TRABALHO DE GRADUAÇÃO

# DETECÇÃO DE *REMOTE ACCESS TROJAN* COM FERRAMENTAS DE *BIG DATA*

**Felipe Edson Pereira Tamanqueira**

**Brasília, Dezembro de 2017**

**UNIVERSIDADE DE BRASÍLIA**

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASÍLIA
Faculdade de Tecnologia

TRABALHO DE GRADUAÇÃO

# DETECÇÃO DE *REMOTE ACCESS TROJAN* COM FERRAMENTAS DE *BIG DATA*

**Felipe Edson Pereira Tamanqueira**

Relatório submetido ao Departamento de Engenharia

Elétrica, como requisito parcial para obtenção

do grau de Engenheiro de Redes de Comunicação

Banca Examinadora

Prof. Flávio Elias Gomes de Deus, ENE/UnB
Orientador _____

Prof. Robson de Oliveira Albuquerque, ENE/UnB
Examinador Interno _____

Prof. Valério Aymoré Martins, ENE/UnB
Examinador Interno _____

# Dedicatória

*Dedico este trabalho a minha família e àqueles que se empenham em combater crimes virtuais, melhorando assim a experiência dos usuários de redes de computadores.*

*Felipe Edson Pereira Tamanqueira*

# Agradecimentos

*Agradeço a minha esposa Bárbara pela paciência e apoio durante todos os momentos que precisei para realizar este projeto. Agradeço ao professor Flavio Elias pela oportunidade de desenvolver este trabalho e pelo esforço em me orientar da melhor forma possível. Agradeço ao brigadeiro Mesquita e aos amigos que me ajudaram e incentivaram muito a continuar o curso de engenharia, durante o tempo que trabalhei na Força Áerea em Brasília. Agradeço ao Juvenal e ao Alex, do Centro de Informática da UNB, que propiciaram a continuidade de meus estudos, sempre me apoiando na medida do possível.*

*Felipe Edson Pereira Tamanqueira*

# RESUMO

*Remote Access Trojan (RAT) é um tipo de malware que permite ao atacante acesso com privilégios administrativos à máquina infectada, sendo capaz de executar diversas ações, tais como roubo de informações pessoais e monitoramento das atividades da vítima. Devido ao crescente surgimento de códigos maliciosos, há a necessidade de se criar ferramentas e métodos de prevenção que impeçam novos ataques. Este trabalho apresenta uma proposta de sistema de detecção de RAT, baseado em anomalia de comportamento, cujo funcionamento não precisa da assinatura do malware para ser eficaz. A modelagem do comportamento das conexões é feita com os algoritmos de machine learning K-means, o qual fornece a identificação de grupos de conexões similares, a partir de características das conexões que apresentam valores próximos, e o Random Forest que é utilizado para classificar os fluxos de conexões em normal ou em RAT, segundo os valores extraídos das características das conexões e grupos identificados na captura do tráfego. E para que seja aplicável a qualquer cenário de redes de computadores, foram utilizadas ferramentas de big data, por serem capazes de suportar grandes fluxos de dados. O framework utilizado nesta aplicação é o Apache Spark por que realiza processamento em memória e pode ser aplicado como sistema distribuído, tornando a implementação escalável e com elevado desempenho de execução. Os requisitos de desenvolvimento deste projeto foram: tempo máximo de execução de cinco minutos, baixo custo computacional e acurácia do modelo similar ou superior aos trabalhos relacionados. Assim, após a implementação do projeto, foi possível detectar conexões de novos RAT, apenas com as características de conexões dessa classe de malware, obtendo acurácia de 95%, taxa de falso negativo de 9.9% e taxa de falso positivo de 4.7%. O tempo de execução da detecção se aproximou de um minuto no modo standalone, ao utilizar 512 MB de memória RAM DDR3 1333 Mhz e oito núcleos de processadores Intel(R) Xeon(R) CPU E5-2650 v2 2.6 Ghz, aplicado em uma captura de 72 GB com diversos tipos de aplicações.*

***Palavras-chave:*** *RAT, malware, Apache Spark, Tshark, Sistema de detecção*

# ABSTRACT

Remote Access Trojan (RAT) is a type of malware that allows the attacker access with administrative privileges to the infected machine, being able to perform several actions, such as stealing personal information and monitoring the victim's activities. Due to the increasing appearance of malicious code, there is a need to create prevention tools and methods that prevent further attacks. This work presents a proposal of a RAT detection system, based on behavioral anomaly, whose operation does not need the signature of the malware to be effective. The modeling of the behavior of the connections is done with the machine learning algorithms K-means, which provides the identification of groups of similar connections, which the characteristics of the connections that present close values, and the Random Forest that is used to classify the flows of connections in normal or in RAT, according to the values extracted of the characteristics of the connections and groups identified in the capture of the traffic. And to be applicable to any computer network scenario, big data tools were used because they were able to support large data flows. The framework used in this application is Apache Spark because it performs memory processing and can be applied as a distributed system, making the implementation scalable and with high execution performance. The development requirements of this project were: maximum execution time of five minutes, low computational cost and accuracy of the model similar or superior to the related works. After the implementation of the project, it was possible to detect new RAT connections, only with the connection characteristics of this class of malware, obtaining 95% accuracy, false negative rate of 9.9% and false positive rate of 4.7%. Detection runtime approached one minute in standalone mode, using 512 MB of DDR3 1333 Mhz RAM and eight Intel (R) Xeon (R) CPU cores E5-2650 v2 2.6 Ghz, applied in one capture of 72 GB that contains several types of applications.

***Keywords:*** *RAT, malware, Apache Spark, Tshark, Detection system*

# SUMÁRIO

# LISTA DE FIGURAS

# LISTA DE TABELAS

# LISTA DE ABREVIATURAS

## Acrônimos

| | |
|---|---|
| API | *Application Programming Interface* |
| ASF | Apache *Software Foundation* |
| BSD | Berkeley *Software Distribution* |
| CLI | *Command Line Interface* |
| CSV | *Comma-Separated Values* |
| DCE | *Distributed Computing Environment* |
| DPI | *Deep Packet Inspection* |
| ETL | *Extract-transform-load* |
| FIFO | *First In, First Out* |
| GB | Gigabyte |
| GHZ | Gigahertz |
| HDFS | *Hadoop Distributed File System* |
| HTTP | *Hypertext Transfer Protocol* |
| IC3 | *Internet Crime Complaint Center* |
| IDS | *Intrusion Detection System* |
| IEEE | *Institute of Electrical and Electronics Engineers* |
| IP | *Internet Protocol* |
| IPS | *Intrusion Prevention System* |
| JAR | Java ARchive |
| JSON | JavaScript *Object Notation* |
| JVM | Java Virtual *Machine* |
| MSRPC | *Microsoft implementation of the* DCE/RPC *standard* |
| P2P | *Peer-to-Peer* |
| PCAP | *Packet CAPture* |
| PUP | *Potentially Unwanted Programs* |
| RAT | *Remote Access Trojan* |
| RDD | *Resilient Distributed Dataset* |
| RDP | *Remote Desktop Protocol* |
| RPC | *Remote Procedure Call* |
| RPM | Rotações Por Minuto |
| RSA | *Royal Society for the encouragement of Arts, Manufactures and Commerce* |
| SCTP | *Stream Control Transmission Protocol* |
| SMB | *Server Message Block* |
| SQL | *Structured Query Language* |
| TB | Terabyte |
| TCP | *Transmission Control Protocol* |
| UDP | *User Datagram Protocol* |
| URL | *Uniform Resource Locator* |
| WMI | *Windows Management Instrumentation* |

# Capítulo 1

# Introdução

De acordo com o relatório realizado pelo *Internet Crime Complaint Center* (2017), em 2016, foi recebido um total de 298.728 mil reclamações de perdas financeiras relacionadas a crimes virtuais, resultando em um impacto superior a $1.3 bilhões.

Os crimes que tiveram maior frequência foram: não pagamento ou recebimento de produtos e serviços, violação de dados pessoais e golpes de pagamentos. Somente com violação de dados pessoais e corporativos o prejuízo calculado foi de $155.009,142 milhões, representando um custo expressivo para vítimas. A estimativa desse centro é que apenas 15% dos crimes virtuais são reportados, logo esse impacto é muito maior. A figura 1.1 exibe a quantidade de reclamações *versus* o total de perdas associadas a esses crimes entre 2010 e 2016.

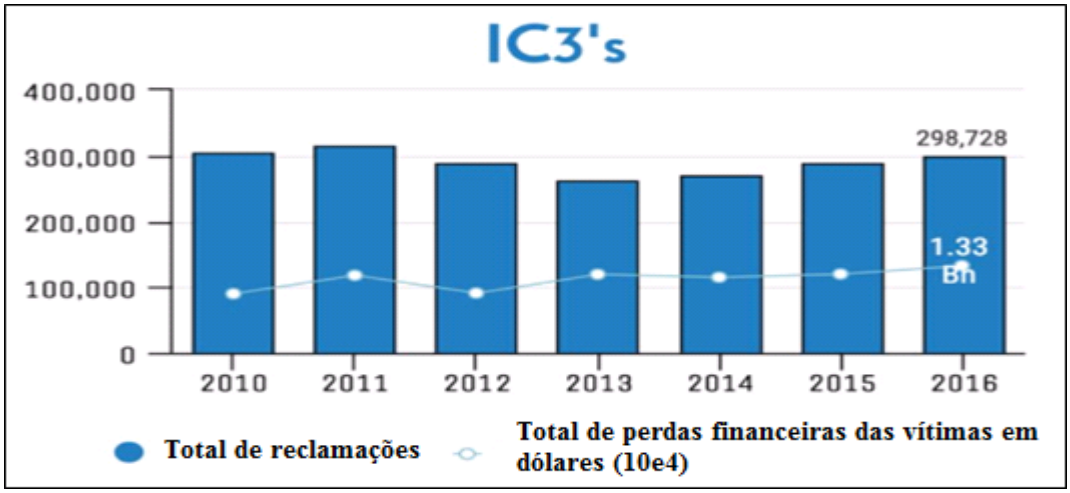



Figura 1.1: Reclamações e perdas financeiras no período de 2010 a 2016 (IC3, 2017)

Além disso, um estudo elaborado pela Verizon Enterprise Solutions (2017), apontou que cerca de 75% das violações de dados são causadas por agentes externos a corporações, estando 51% com participação de organizações criminosas e 18% com participação de entes governamentais. Dentre todas as violações, 62% estão relacionadas a ataques de hackers, das quais 51% utilizam algum tipo de malware para alcançar seu objetivo. Desse total de códigos maliciosos, 66% são instalados por meio de anexos em e-mail. Por fim, em 73% dos casos há motivação financeira.

Da mesma forma, há uma pesquisa realizada por Vaidya (2015), a qual apresenta os principais ataques cibernéticos no período de 2001 a 2013, analisando suas motivações, alvos e técnicas utilizadas.

Primeiramente, ele classifica os ataques em não direcionados, isto é, aqueles que não possuem um alvo específico, têm grande capacidade de disseminação e, geralmente, aproveitam-se das vulnerabilidades de protocolos de comunicação. Em

seguida, identifica também ataques direcionados a certo alvo, tais como nações, segurança nacional, organizações, pessoas, entre outros.

Vaidya (2015) relata que técnicas de *phishing* são os mecanismos mais utilizados para propagar *malwares* que exploram vulnerabilidades de sistemas. Ainda descreve que os ataques de negação de serviço são menos prejudiciais do que os ataques de roubo de informação, pois a captura de dados agrega mais valor ao atacante, seja por conter informação sensível, como, por exemplo, segredo industrial, ou por adquirir credenciais e informação bancárias, podendo até mesmo sequestrar os dados para uma futura negociação com a vítima. Por isso, o furto de dados é o principal motivador para ataques cibernéticos, seguido de espionagem, pois ambos oferecem benefícios parecidos.

Outro resultado importante dessa pesquisa foi o de que a maioria dos ataques ocorreu devido a vulnerabilidades conhecidas, e que possuíam atualizações disponíveis capazes de prevení-las. Assim, grande parte dos ataques poderia ter sido evitada apenas com política de segurança proativa, no caso de organizações, verificando as vulnerabilidades recentes e obrigando a atualização dos ativos, e no caso de pessoas físicas, mantendo seus softwares atualizados.

Atualmente, há um caso bem recente desse tipo de falha de segurança, o *Wannacry*, que é um *ransomware* que explora a vulnerabilidade do protocolo de compartilhamento *Server Message Block* (SMB) do *Windows*, cuja prevenção é uma simples atualização.

Em relação aos *malwares*, Benzmüller (2017) analisou o quantitativo de novos tipos de *malwares* que são identificados diariamente. Para realizar esse estudo, foi considerado apenas os tipos representados por assinaturas, pois são constituídos de códigos similares e possuem pouca variação. O objetivo desse tipo de contagem é reduzir a duplicidade gerada quando se levam em considerações arquivos infectados e malwares polimórficos, como é feito pelo site AV-*Test*.

O resultado dessa pesquisa pode ser observado na figura 1.2, a qual apresenta, em milhões, a evolução de novos tipos de *malwares* de 2007 até o primeiro semestre de 2017, incluindo previsão até o final do ano. Pode-se notar que há um grande crescimento da criação de códigos maliciosos, obtendo-se uma curva quase exponencial, principalmente nos últimos quatro anos.

Dando continuidade à estatística, o autor relata que diariamente mais de 27.000 novos tipos de *malwares* são identificados, e consequentemente uma média de um a cada 3.2 segundos.

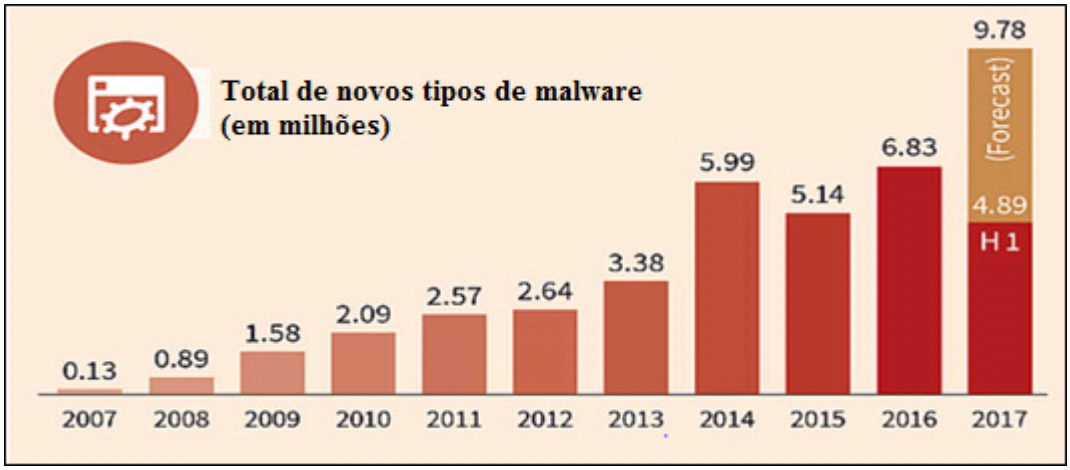



Figura 1.2: Total de novos tipos de *malwares* no período de 2007 a 2017 (Benzmüller, 2017)

Segundo Benzmüller (2017), as classes de *malwares* não mudaram muito ao longo dos anos. No que tange à proporção de códigos maliciosos criados em cada classe, tem-se o *trojan horse* com a maior representatividade, seguido do *adware* e do *Potentially Unwanted Programs* (PUP). O número de *ransomwares*, por sua vez, também aumentou, porém de forma lenta, com apenas 0.1% de aumento. Dentre as plataformas alvos desses códigos maliciosos, o *Windows* ainda é o mais visado.

Agora, usando como base ataques detectados no primeiro semestre de 2017, tem-se na tabela 1.1 a incidência dos dez primeiros tipos de *malwares*, e a quantidade de ataques relativos a mil usuários.

Tabela 1.1: Os dez tipos de *malware* mais utilizados em 1°/2017 (Adaptado de Benzmüller, 2017)

| Rank | Tipos de Malware | % | # por 1000 usuários |
|---|---|---|---|
| 1 | Trojan.BAT.Poweliks.Gen | 25,20% | 3252 |
| 2 | JS:Trojan.JS.Agent.RB | 3,50% | 452 |
| 3 | Exploit.Poweliks.Gen.4 | 2,90% | 379 |
| 4 | Win32.Trojan.Binder.A | 1,50% | 199 |
| 5 | Gen:Variant.Binder.1 | 1,50% | 198 |
| 6 | JS:Trojan.Cryxos.261 | 1,00% | 126 |
| 7 | Script.Trojan.Redirector.BA | 0,90% | 119 |
| 8 | Exploit.JAVA.CVE-2013-0431.G | 0,90% | 110 |
| 9 | Trojan.Snifula.Gen.1 | 0,70% | 89 |
| 10 | Win32.Worm.Autorun.A@gen | 0,50% | 64 |

## 1.1 Problema

Diante do exposto, percebe-se a importância de se projetar métodos e sistemas que impeçam esses vetores de ataques, e, para isso há diversos tipos de soluções: *Firewall*, *Intrusion Prevention System* (IPS), *Intrusion Detection System* (IDS), antivírus, e outros, cada um mitigando riscos de segurança da informação de um jeito. Entretanto, vale ressaltar, que a maioria desses sistemas tem em comum a detecção de ataques baseada em um padrão específico de comportamento do

*malware*, chamado de assinatura, que é armazenada em banco de dados e comparada com o fluxo de dados analisado, com intuito de identificar o *malware* e impedir a invasão do atacante ou atividade maliciosa.Porém não são capazes de detectar imediatamente novas versões de *malwares*, sem assinatura prévia.

Ao passo que se aumenta as facilidades com novos recursos, aumenta-se a complexidade, e consequentemente aumentam-se também as falhas de projeto de *software*, gerando novas maneiras e ferramentas de se efetuar ataques.

Com o intuito de evitar novos ataques, foram elaborados métodos de detecção de intrusão, baseados em anomalias de comportamento, que não consideram a assinatura, a fim de não somente identificar novos tipos de *malwares*, mas também detectar ofuscações de ameaças conhecidas.

Benzmüller (2017) afirma que os *trojans* são as classes de *malware* que possuem maior ocorrência de ataques atualmente, portanto são um tipo de *malware* com grande relevância de estudo. Além disso, o roubo de informação representa alto impacto financeiro para vítimas, e na literatura tem-se um tipo de subclasse de *trojan* utilizada para esse fim, o *Remote Access Trojan* (RAT).

De acordo com o Wu et al. (2017), os métodos de detecção baseados em anomalias ainda não foram implementados de forma relevante em ambientes reais. Além disso, há poucos estudos da aplicação desse método em sessões de RAT, limitando-se a exemplos de execução. Este autor indica que o *machine learning* é a principal técnica para identificar esse *malware* e sua eficiência depende da especificação das características das sessões de RAT.

Segundo Jiang e Omote (2015), RAT é um tipo de *spyware* usado para invadir estações de trabalho por meio de ataques direcionados, a fim de monitorar e controlar o computador remotamente, esperando uma oportunidade para roubar informações confidenciais. Entende-se *spyware* como “um programa projetado para monitorar as atividades de um sistema e enviar as informações coletadas para terceiros.” (CERT BR, 2012, p.27).

A exemplo de RAT, há o caso relatado pela RSA *Research* (2015), no qual foi descoberto um tipo de *malware zero day*, denominado GlassRAT, que foi empregado em um popular *software* na China, durante anos sem detecção. O alvo desse RAT eram organizações geopolíticas na região da Ásia-Pacífico.

## 1.2 Objetivo geral

Estudar a capacidade de detecção de um sistema baseado em anomalia de comportamento com ferramentas de *big data*, a fim de identificar atividade maliciosa de RAT em redes internas.

## 1.3 Objetivos específicos

- Caracterizar os comportamentos dos RAT
- Propor um modelo de sistema de detecção de RAT
- Apontar tecnologias capazes de serem usadas na detecção de intrusão

- Verificar requisitos necessários para implementação do modelo proposto
- Analisar a eficiência do modelo com diversas famílias de RAT

## 1.4 Justificativa

O surgimento de novos tipos de *malwares*, aliado à constante perda de recursos financeiros com violação de dados, bem como espionagem, demandam o aprimoramento dos sistemas de detecção e prevenção de invasão.

Assim, este projeto busca elaborar um método baseado em anomalia de comportamento capaz de identificar qualquer ameaça de RAT em rede interna, utilizando ferramentas *big data*. Essas ferramentas possuem elevada capacidade de processamento de grandes dados, e quando aplicadas à análise de tráfego de redes de computadores, reduzem a complexidade computacional ocasionada pelo grande fluxo de dados gerados no cotidiano.

Portanto, foi planejado um modelo de sistema de detecção capaz de capturar continuamente o tráfego de uma rede interna e extrair informações dos pacotes de forma automatizada, além disso, foi necessário avaliar a possibilidade de redução do custo computacional de processamento das capturas, para essas atividades foi utilizada a ferramenta Tshark e seus componentes. Para implementar a parte de reconhecimento e classificação do comportamento das conexões foram utilizados os módulos SQL e MLlib do Apache Spark, empregando a combinação dos algoritmos de aprendizagem de máquina K-means e Random Forest.

## 1.5 Estrutura do trabalho

O presente estudo está dividido em cinco capítulos. No primeiro capítulo, foram apresentados os objetivos do projeto, assim como o problema a ser estudado, de forma contextualizada.

No segundo capítulo, será realizada revisão da literatura, a fim de mostrar os principais trabalhos relacionados ao tema, e apresentar conceitos necessários para o entendimento da elaboração do projeto.

O terceiro capítulo consiste em relatar o modelo de sistema de detecção sugerido para o problema proposto. O desenvolvimento da descrição do projeto se dará em duas etapas: a primeira exibirá a arquitetura idealizada com as ferramentas empregadas, ilustrando os métodos adotados; já na segunda etapa será feita uma análise de desempenho das ferramentas, com intuito de identificar o requisito mínimo para reprodução do projeto em ambiente real.

No quarto capítulo, o sistema de detecção será validado em dois estudos de casos: no primeiro, a detecção é aplicada em ambiente real com capturas de RAT criadas especificamente para o projeto. Já no segundo caso, as capturas de RAT utilizadas são obtidas na *internet*, sendo disponibilizadas por terceiros. Ao final será comparada a acurácia em cada cenário.

Por fim, o quinto capítulo contém a conclusão e os trabalhos futuros.

# Capítulo 2

# Fundamentação teórica

Este capítulo apresenta os principais trabalhos relacionados ao tema, fornecendo base teórica para compreensão do problema. E expõe os conceitos e ferramentas necessárias para implementar o projeto.

## 2.1 Conceitos

Dado que foram abordadas algumas ferramentas e conceitos na seção anterior, faz-se necessário realizar uma breve revisão da literatura, com o intuito de apresentar o funcionamento dos algoritmos de *machine learning* e do *framework* aplicados no projeto. Também serão revistos os cabeçalhos dos protocolos do TCP/IP envolvidos na aplicação do modelo de detecção, a fim de esclarecer a adoção do critério de truncamento dos pacotes que será utilizado.

### 2.1.1 Arquitetura TCP/IP

Segundo Forouzan (2007), o conjunto de protocolos TCP/IP possui quatro camadas, denominadas como: *host*-rede, *internet*, transporte e aplicação, de acordo com a figura 2.1. Porém esse autor, para fins práticos considera a camada *host*-rede como a combinação das camadas física e enlace do modelo OSI.

Nota-se na figura 2.1, que cada camada contém protocolos específicos que desempenham determinadas funções no processo de comunicação. Conforme esse autor, a camada *host*-rede não possui nenhum protocolo particular, suportando todos os protocolos desenvolvidos para funcionar nessa camada.

Forouzan (2007) também definiu que os principais protocolos utilizados em transmissão de dados na camada de internet é o *Internet Protocol* (IP), e na camada de transporte são o TCP, o *Stream Control Transmission Protocol* (SCTP) e o UDP.

O TCP/IP constitui um modelo hierárquico, o qual protocolos de nível superior são suportados por um ou mais protocolos inferiores. O IP é um protocolo sem conexão e não confiável, desenvolvido para oferecer serviço com melhor esforço possível, pois não faz verificação ou correção de erro. Todavia o protocolo TCP fornece serviço orientado a conexão, e implementa mecanismos de controle de fluxo e de erros na camada de transporte.

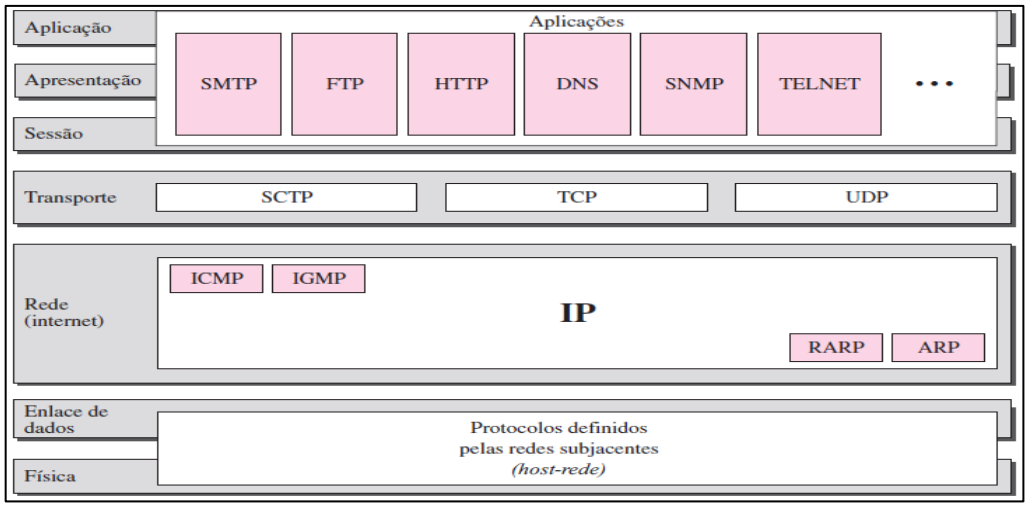



Figura 2.1: Camadas TCP/IP (Forouzan, 2007, p. 43)

De acordo com Forouzan (2007), cada camada pode acrescentar um cabeçalho ou *trailer*, cuja função é adicionar informações que serão usadas pelas camadas durante a comunicação, e, como consequência há o encapsulamento do dado da camada acima.

Conforme Kurose e Ross (2013), em cada camada há dois tipos de campos: campos de cabeçalho, que são as informações adicionais, e um campo de carga útil que geralmente é o dado entregue pela camada superior.

A figura 2.2 ilustra muito bem o processo de comunicação de um *host* para outro, passando por ativos de redes, e mostrando, de forma resumida, qual cabeçalho é lido por cada ativo, e como ocorre o procedimento de encapsulamento.

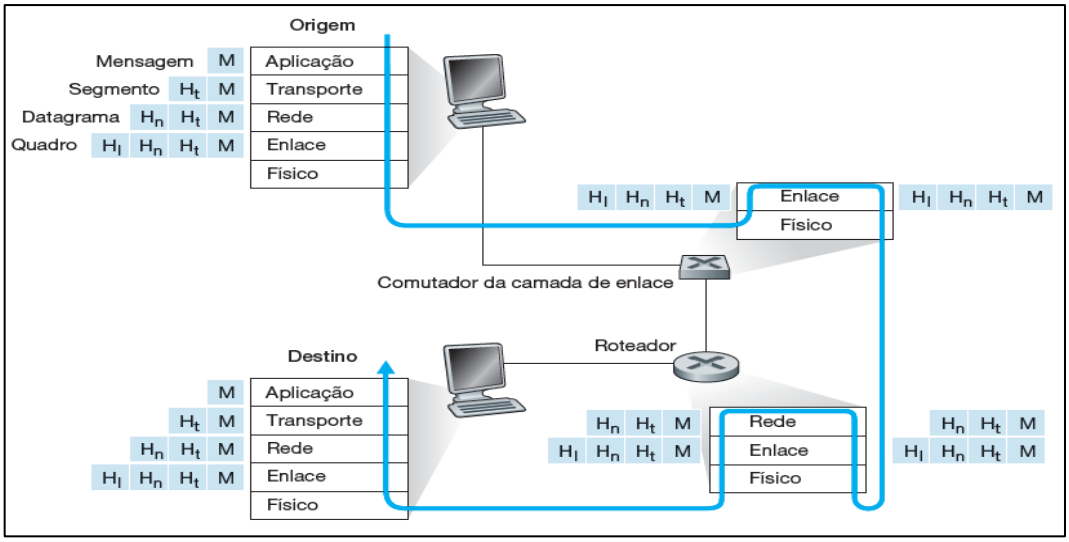



Figura 2.2: Encapsulamento nas camadas (Kurose e Ross, 2013, p.40)

Como foi dito, na camada de enlace, podem ser utilizados diversos protocolos de comunicação, uma vez que dependerá da tecnologia empregada. Em redes de computadores internas, tem-se como padrão a utilização do IEEE 802.3 (ethernet). Na figura 2.3, podem ser vistos todos os campos desse protocolo e os seus respectivos tamanhos.

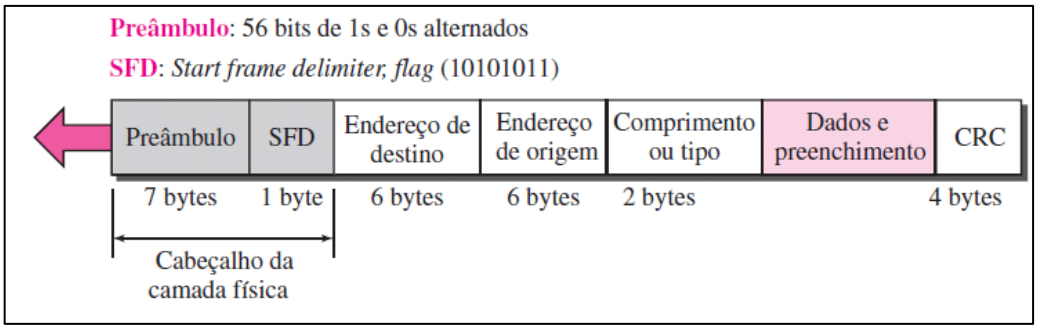



Figura 2.3: Quadro 802.3 (Forouzan, 2007, p. 398)

Conforme Forouzan (2007), os campos preâmbulo e SFD fazem parte da camada física, sendo aquele responsável por alertar a chegada do quadro, possibilitando a sincronização do *clock* de entrada no receptor, enquanto o SFD informa o início do quadro. Assim esses campos não fazem parte do quadro IEEE 802.3.

A figura 2.4, apresenta os tamanhos mínimo e máximo do quadro IEEE 802.3. Segundo Forouzan (2007), o tamanho mínimo do quadro, o qual soma o cabeçalho e a carga útil, deverá ser de 64 bytes. O cabeçalho juntamente com trailer possui 18 bytes, caso a carga útil não tenha 46 bytes para alcançar o tamanho mínimo, será preenchido automaticamente com bits restantes. Além disso, o padrão também define tamanho máximo de bytes, que é o de 1518 bytes, assim a carga útil poderá atingir até 1500 bytes.

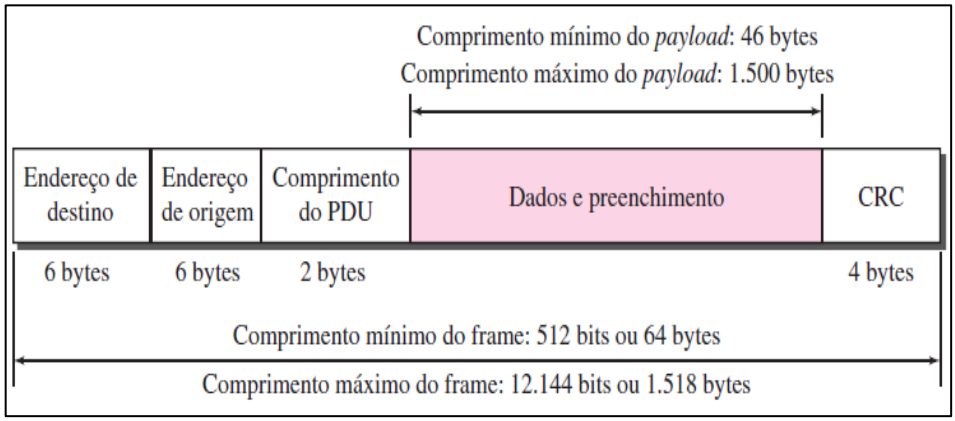



Figura 2.4: Comprimento mínimo e máximo do IEEE 802.3 (Forouzan, 2007, p. 399)

Cabe ressaltar que o endereço de origem e de destino na figura 2.4 se refere ao endereço MAC contido nas interfaces de rede dos *hosts* ou ativos de rede, e que o CRC contém informações para detecção de erro.

Já na camada de rede temos o IP, que em sua versão 4, possui endereçamento de 4 bytes (32 bits). Basicamente, sua função é identificar um equipamento em uma rede de computadores com objetivo de entregar ponto a ponto os dados criados pelas camadas superiores.

Contudo, sabe-se que atualmente o endereçamento com essa quantidade de bits não é mais suficiente para atender a demanda dos usuários. Para isso foi criado o IP versão 6, este ainda está sendo implementado de forma lenta. Portanto o IPV4 é dominante nas redes de computadores, sendo portanto fundamental para a *internet*.

A figura 2.5, demonstra o cabeçalho do IPV4 com seus campos e, de acordo com Forouzan (2007), o tamanho do datagrama IPV4 pode variar de 20 a 60 bytes, pois o campo

opção pode acrescentar 40 bytes. Assim, caso não seja utilizado o campo opção, o tamanho mínimo é de 20 bytes.

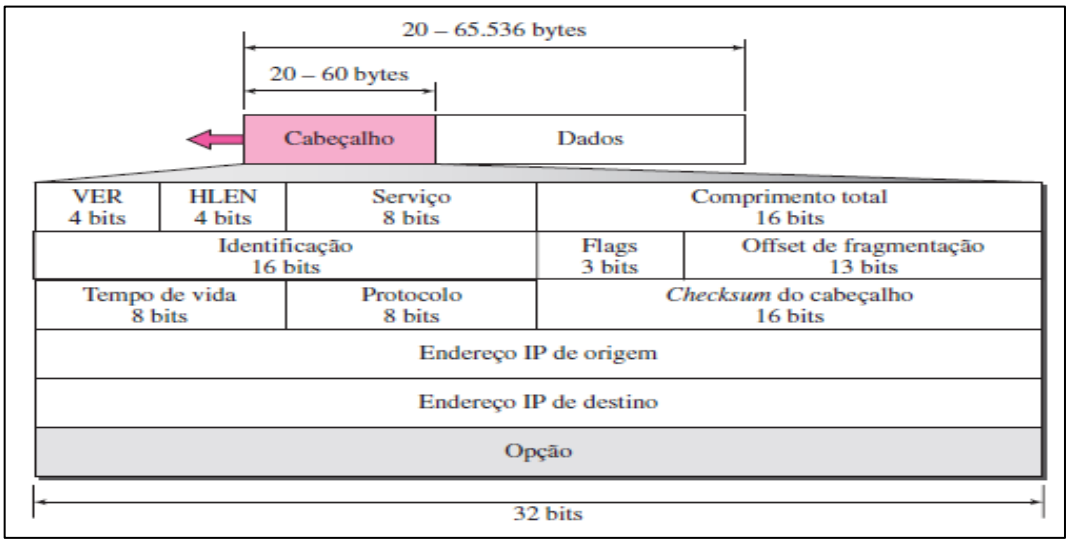



Figura 2.5: Datagrama IPV4 (Forouzan, 2007, p. 583)

Dando prosseguimento aos comentários sobre os protocolos mais importantes do TCP/IP, na camada de transporte, conforme figura 2.6, o segmento do protocolo TCP possui cabeçalho de 20 a 60 bytes, e o seu tamanho é variável de acordo com a utilização do campo opções e preenchimento. Os endereços de porta de origem e destino são números que identificam a porta utilizada pelo programa da camada de aplicação na comunicação. Após o campo reservado, tem-se o campo de controle com seis bits, sendo responsável por implementar controle de fluxo, estabelecimento e encerramento de conexão.

Segundo Kurose e Ross (2013), dentro do campo de controle, a *flag* PUSH, representada como PSH na figura 2.6, é usada para indicar ao destinatário que os dados devem ir para a camada superior imediatamente, logo não deverá aguardar o recebimento de mais dados para executar o envio.

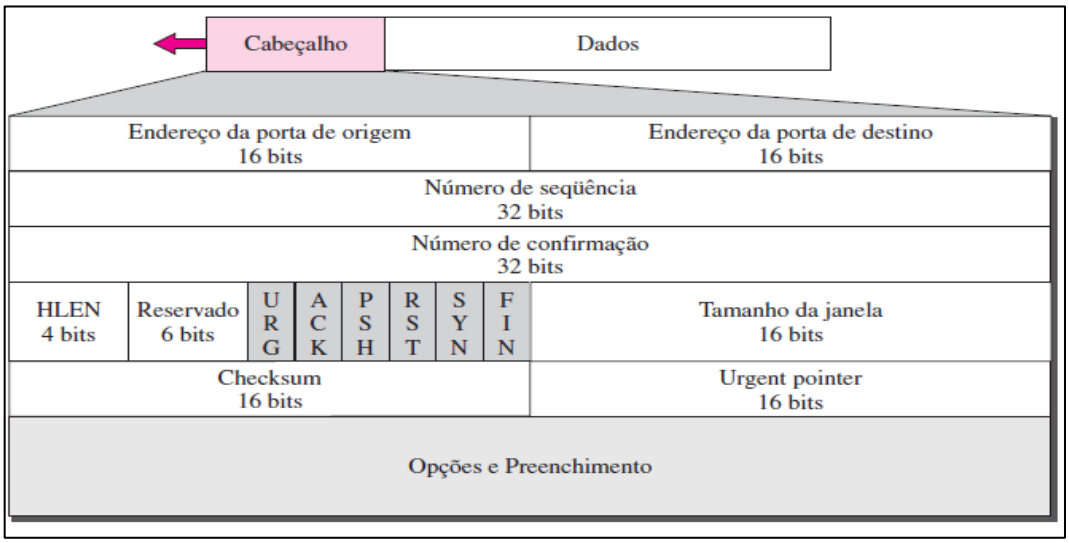



Figura 2.6: Segmento TCP (Forouzan, 2007, p. 721)

## Wireshark

De acordo com Bullock e Parker (2017), o wireshark é uma ferramenta de análise de rede e protocolo, gratuita, que pode ser usada em diversos sistemas operacionais, tais como Windows e Linux. Os pacotes capturados são interpretados e apresentados de forma individual para análise nessa ferramenta.

Esse *software* possui dois modos de execução: interface gráfica e CLI. O primeiro pode ser visto na figura 2.7, esse modo oferece a exibição detalhada dos pacotes e agrupa as informações em camadas, o que facilita a análise.

Além de possibilitar a visualização dos pacotes de forma simples e intuitiva com a interface gráfica, há a capacidade de se utilizar o CLI, seja desenvolvendo *scripts* com o wireshark API para o interpretador Lua ou usando o wireshark modo CLI, denominado Tshark.

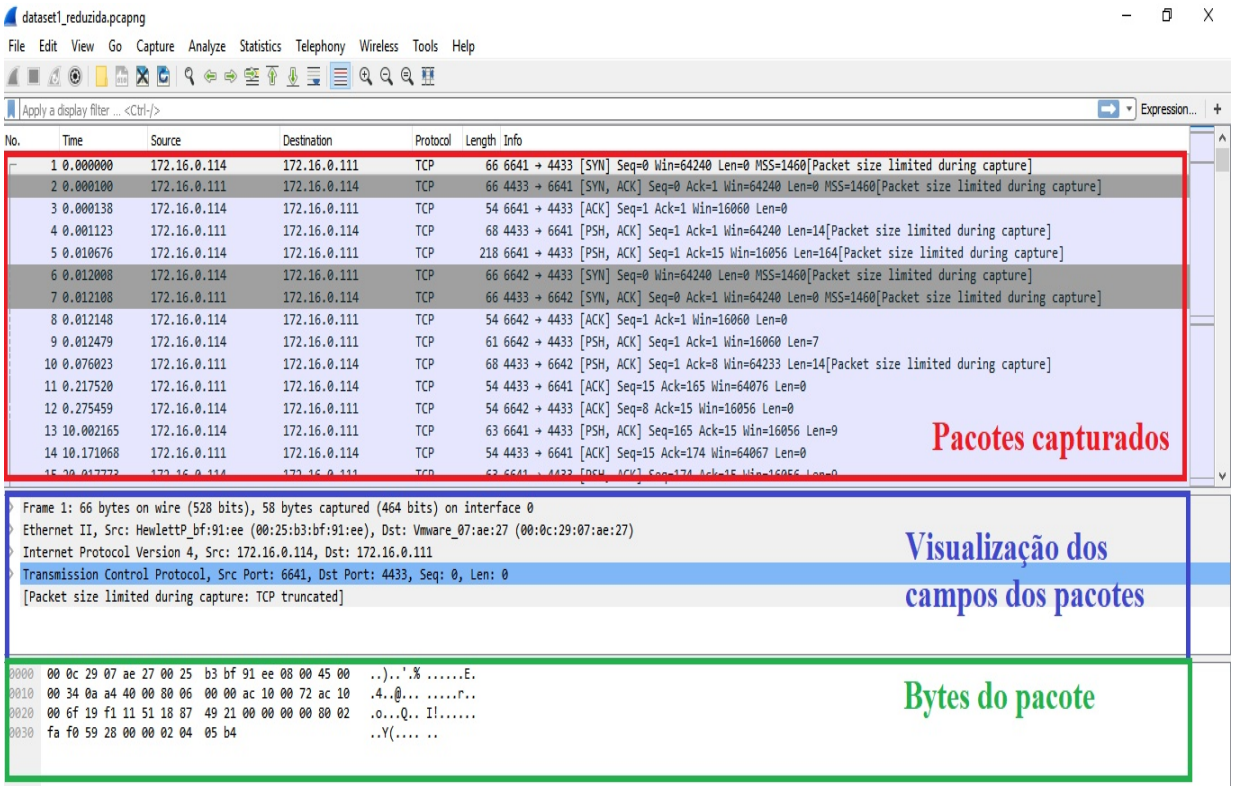



Figura 2.7: Wireshark modo gráfico

Segundo Bullock e Parker (2017), o modo Tshark, figura 2.8, é menos conhecido e altamente subutilizado. Este modo possui basicamente as funcionalidades da ferramenta tcpdump, mas com acréscimo de utilitários do wireshark. Comparando o Tshark com a interface gráfica, aquele é mais eficiente e apropriado para trabalhar com *scripts*.

Além disso, quando se lida com capturas bem grandes, arquivos em gigabytes, a interface gráfica se torna inviável, dependendo dos recursos da máquina que realiza a captura e/ou análise do tráfego. Assim o Tshark é o mais indicado para manipular capturas com grandes fluxos de pacotes de uma rede.

```
localhost:~$ tshark
31 5.064302000 192.168.178.30 -> 173.194.67.103 TCP 74
48231 > http [SYN] Seq=0
   Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=926223
TSecr=0 WS=1024
32 5.074492000 192.168.178.30 -> 194.109.6.66 DNS 75
Standard query 0x56dc  A forums.kali.org
33 5.074987000 192.168.178.30 -> 46.51.197.88 TCP 74
59132 > https [SYN] Seq=0
   Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=926226
TSecr=0 WS=1024
34 5.082801000 192.168.178.30 -> 46.228.47.115 TCP 74
33138 > http [SYN] Seq=0
   Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=926228
TSecr=0 WS=1024
35 5.103958000 192.168.178.30 -> 91.198.174.192 TCP 66
```

Figura 2.8: Wireshark modo CLI – Tshark (Bullock e Parker, 2017, p.82)

Em Wireshark (2017), tem-se a documentação do wireshark no modo CLI, nesse site é possível identificar outras ferramentas que auxiliam na manipulação dos arquivos gerados com extensão PCAP (arquivos que contém os pacotes capturados), tais como editcap, capinfos e mergecap. Há também a documentação do Tshark, descrevendo todas as funcionalidades e como usar os parâmetros, a fim de facilitar a elaboração do *script*.

### 2.1.2 *Machine learning*

Segundo Gates (2017), atualmente há diversas aplicações de algoritmos de aprendizagem de máquina, como por exemplo, sistemas de detecção de fraude de cartão de crédito, sistemas que filtram mensagens spam e mecanismos de pesquisa. Diferente de programas convencionais, esses algoritmos utilizam inteligência artificial humana para aprender com experiências, em vez de seguir instruções sequenciais, podendo adaptar-se e detectar padrões complexos durante a sua execução.

Assim, os algoritmos de *machine learning* aprendem a partir de entradas e fornecem respostas para problemas complexos. Essa entrada pode ser denominada de dado de treinamento que representa a experiência adquirida previamente, enquanto a saída é o resultado da tarefa realizada de detecção de padrões complexos.

Conforme Hartshorn (2016), a maioria dos algoritmos apresentam as seguintes características:

- Inicia a aprendizagem com um conjunto de dados conhecidos, o qual se sabe a resposta para uma pergunta, sendo denominado de dado de treinamento;
- Treina-se o algoritmo com dados de treinamento, com intuito de se obter o modelo.
- Insere outro conjunto de dados, o qual se quer respostas do modelo obtido, denominado dados de testes; e
- O algoritmo fornece como saída a resposta dos dados testados.

De acordo com Sullivan (2017), os algoritmos de *machine learning* podem ser classificados em quatro tipos:

- *Supervised learning* (supervisionados) – o algortimo é treinado com amostras de dados contendo entradas e suas saídas esperadas;
- *Unsupervised learning* (não supervisionados) – neste caso, não há treinamento com valores de saída predefinidos, cabe ao algoritmo definir a saída de acordo com as entradas, como por exemplo, agrupar dados semelhantes;
- *Semi-supervised learning* (semi-supervisionados) – é a utilização combinada de algoritmos supervisionados e não supervisionados; e
- *Reinforcement learning* (por reforço) – são dadas recompensas ao algoritmos, conforme sequência de ações, geralmente é aplicado em inteligência artificial.

#### 2.2.3.1 Algoritmo *K-means*

Segundo Nicolas (2014), o objetivo do uso de algoritmos de aprendizagem não supervisionada é descobrir padrões regulares e irregulares em um conjunto de observação. Há inúmeros algoritmos deste tipo, cada um sendo apropriado para um determinado tipo de característica.

Problemas envolvendo grande conjunto de dados se tornam bastante difícil avaliar a independência entre as caraterísticas selecionadas do objeto de estudo. A fim de otimizar recursos computacionais, tem-se como objetivo reduzir conjuntos de dados contínuos, infinitos ou muito grandes em pequenos grupos que compartilham alguns atributos em comum.

Conforme Nicolas (2014), essa divisão em pequenos grupos é chamada de *vector quantization*, e o seu principal benefício é a análise utilizando o conjunto representativo de um grupo, o que é mais simples do que a análise de todo o conjunto de dados.

Outra definição abordada pelo autor é o *clustering*, que é uma forma de *vector quantization* aplicada em conjunto com conceitos de distância ou similaridade com objetivo de gerar grupos, denominados *clusters*.

O K-*means clustering* é um algoritmo de agrupamento interativo, conforme figura 2.9, que elege um representante de cada *cluster*, conhecido como centróide. Em seguida, os demais dados são comparados aos centróides, a fim de verificar a semelhança para agrupar os dados. Há diversas formas de mensurar a similaridade, as principais são:

- Distância Manhattan;
- Distância Euclidiana*;* e
- Cosseno de valores observados.

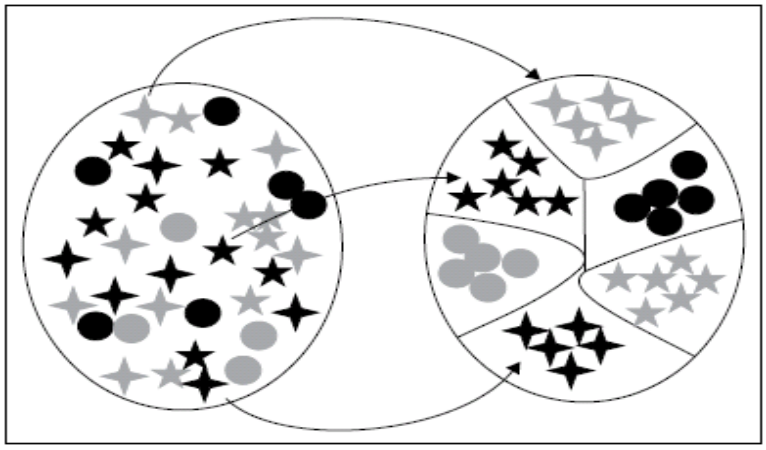


Figura 2.9: Exemplo de agrupamento de dados similares (Nicolas, 2014, p.100)

A distância euclidiana é definida como o quadrado da distância entre dois vetores $x_i$ e $y_i$, conforme equação 2.

$$d(x,y) = \sum(x_i - y_i)^2 \qquad \text{Eq. (2)}$$

O K-*means* tem como principal vantagem a simplicidade, considerando K *clusters* $C_k$ com centróides $m_k$, e de acordo com a equação 3, o objetivo do algoritmo é minimizar o erro total gerado pelo cálculo de similaridade empregado.

$$arg\ min_{C_k} = \sum_1^K \sum_{x_i \epsilon C_k} d(x_i, m_k) \qquad \text{Eq. (3)}$$

De acordo com Nicolas (2014), para utilizar o K-*means* é necessário especificar o número de *clusters* (K) que se deseja agrupar os dados, definir a técnica de cálculo da similaridade e determinar o número máximo de interações do algoritmo.

As interações são utilizadas para a convergência do algoritmo, com intuito de se obter valores confiáveis, e terminam quando as verificações de similaridade não são mais necessárias, ou seja, não há mais reatribuição de um objeto de observação em um grupo.

Uma modelagem com elevados números de características (N) de entrada requer elevado número de objetos de observação para que os *clusters* tenham maior robustez.

Um conjunto de dados com menos de 50 objetos, produz modelo com alta polarização e afeta a verificação de similaridade, esse autor apresenta uma regra empírica para determinar o valor máximo de objetos para os dados de treinamento (n), devendo ser $n < K \cdot N$.

Segundo Duvvuri e Singhal (2016), o K-*means* apresenta como desvantagens: utilizar apenas números como valores das características de entrada, e necessita colocá-los na mesma escala.

### 2.2.3.2 Algoritmo *Random Forest*

Conforme Duvvuri e Singhal (2016), *decision tree* é um algoritmo de aprendizagem supervisionada que pode ser usado tanto para classificação quanto para regressão.

A estrutura do *decision tree* é definida com um nó acima e suas folhas abaixo como ramificações, de acordo com a figura 2.10. Existem diversos algoritmos para dividir o conjunto de dados em segmentos que possuam semelhança.

Essa divisão é feita com base em ganho de informação, que mede o quanto um atributo separa os dados de treinamento da classe alvo. Assim, a primeira divisão dos nós é representada pela característica que oferece o maior ganho de informação, tornando-se o nó raiz.

O ganho de informação de um nó é definido pela diferença entre a impureza do nó pai e a soma ponderada da impureza dos nós filhos (folhas).

Percebe-se ainda que cada nó representa uma característica definida como entrada para o algoritmo, a verificação desse atributo é usado para designar a folha adequada para o objeto avaliado.

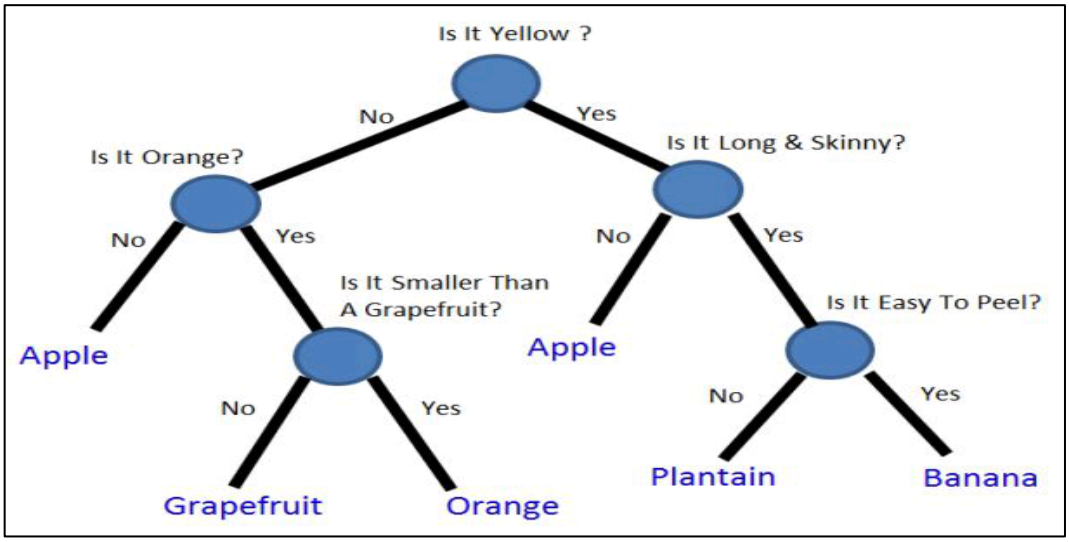



Figura 2.10: Exemplo de *decision tree* (Hartshorn, 2016, p.12)

Segundo este autor, *decision tree* não necessita de normalização ou transformação dos dados de entrada. A impureza é interpretada como medida de homogeneidade e critério para particionamento recursivo, sendo um parâmetro baseado em probabilidade. Os principais cálculos aplicados à impureza são:

- Gini i*ndex*; e
- Entropia

O Gini *index*, equação 4, é empregado geralmente em características com valores contínuos, e caso a entrada seja uma categoria, será assumido que os atributos são contínuos, gerando transformação em valores contínuos. A divisão com Gini *index*, faz com que o nó filho ser mais puro do que o nó pai.

$$\text{Gini } index = 1 - \sum_1^j (p_j)^2 \qquad \text{Eq. (4)}$$

De acordo com Hartshorn (2016), $p_j$ é a probabilidade de se ter uma determinada classe de dado em um conjunto de dados, e j é a quantidade de classes. Tem-se o melhor valor de impureza, igual a zero, quando há somente uma classe de dados, o que resulta $p_j$ igual 1 e j igual a 1.

Segundo Duvvuri e Singhal (2016), métodos que usam mútiplos algoritmos de aprendizado melhoram a precisão do modelo preditivo, porém pode requerer maior recurso computacional e aumentar a complexidade, dificultando também a compreensão dos resultados.

O *Random Forest* é uma técnica que aplica um conjunto de *decision tree*, seu funcionamento consiste em dividir os conjuntos de dados de entrada em dados de treinamento para várias *decision tree* separadas, figura 2.11, e em seguida, os resultados de cada *decision tree* são combinados para gerar a classificação do *Random Forest*.

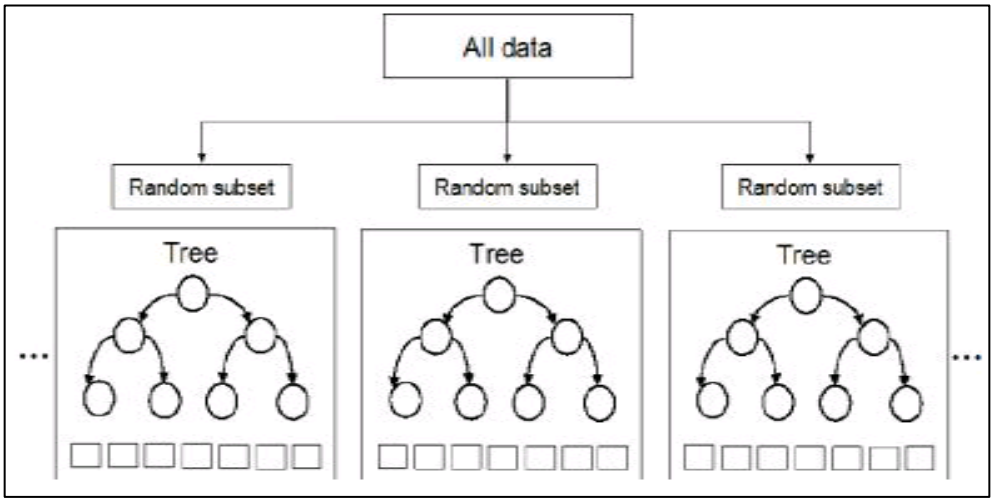



Figura 2.11: Funcionamento do *Random Forest* (Sullivan, 2017, p.77)

Conforme Sullivan (2017), para classificar os objetos de entrada, cada uma das *decision tree* apresenta uma classificação, que é conhecida como votação para a classe semelhante. O *Random Forest* escolhe a classe que possui a maioria de votos gerados pelo total de *decision tree*.

### 2.1.3 Apache Spark

Segundo Aven (2016), atualmente o Spark é o projeto *open source* da ASF mais ativo, e um dos maiores projetos de *big data* de todos os tempos. Esse *framework* é um projeto de processamento de dados distribuído, sendo iniciado em 2009 pelo Berkeley RAD Lab, sob licença BSD. E em 2014 foi incorporado pela ASF, tornando-se um dos seus projetos com mais importantes.

O Spark é codificado na linguagem Scala que é suportada pelo JVM e Java runtime, o que favorece sua aplicação em múltiplas plataformas, tais como Linux e Windows.

De acordo com Frampton (2015), o Spark é um sistema altamente escalável de análise de dados distribuídos em memória, que permite o desenvolvimento de aplicativos nas linguagens de programação Java, Scala, Python e R. Possui quatro módulos principais independentes: SQL, MLlib, GraphX e *Streaming*, conforme figura 2.12, a qual a partir da terceira fileira são exemplos de ferramentas externas que podem se integradas ao Spark.

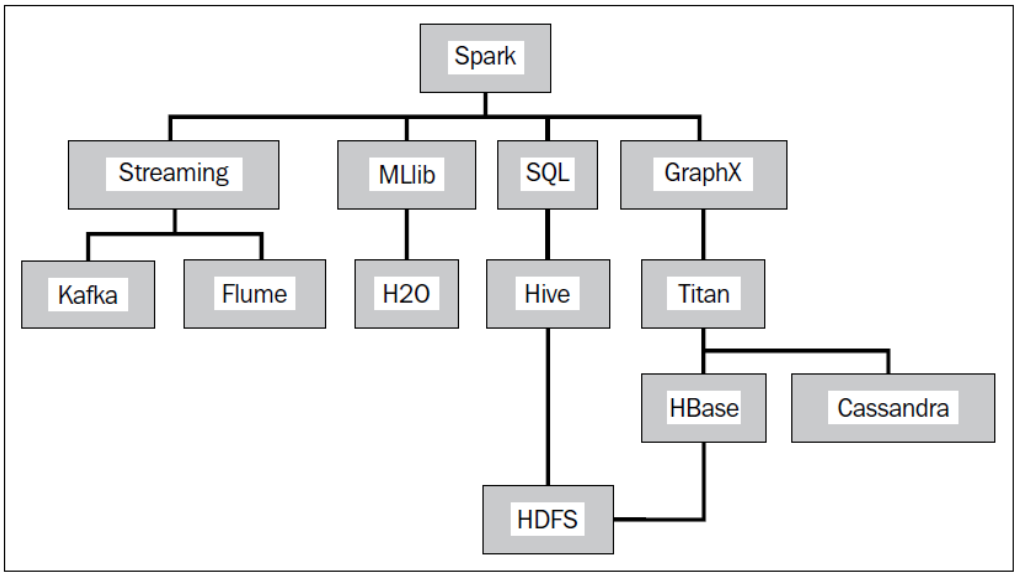



Figura 2.12: Arquitetura Apache Spark (Frampton, 2015, p.1)

Dentre esses módulos, o MLlib oferece funcionalidades para criação de aplicações que utilizam *machine learning*, e o SQL usa, a partir da versão 1.3 do Spark, o conceito de *data frame*, o que permite o processamento de dados em forma tabular, podendo utilizar funções tabulares, tais como *select*, *filter*, groupby.

Este módulo suporta integração com ferramentas externas e consegue armazenar os dados nos formatos CSV, Parquet e JSON, os quais são muitos utilizados em *big data*.

Frampton (2015) ilustra como funciona a gerência de *clusters* do Apache Spark na figura 2.13, de forma geral, há a possibilidade de se utilizar diversos tipos de gerenciadores de *clusters* com o Spark, como por exemplo, o Yarn e o Mesos. Percebe-se também a arquitetura mestre e escravo implementada quando se utiliza os modelos distribuídos.

O Spark *context* pode ser configurado dentro da classe implementada no projeto da aplicação ou por meio da Spark URL, este podendo ser acessado pelo endereço Spark://<hostname>:7077. O Spark *context* se conecta ao gerenciador de *cluster* implementado para executar um conjunto de tarefas, o qual tem a incumbência de alocar os recursos necessários em todos os nós para o Spark.

Além disso, o gerenciador atribui aos nós cópias do arquivo JAR da aplicação que deverá ser executada, com intuito de distribuir a execução da aplicação, e por fim aloca as tarefas para cada nó.

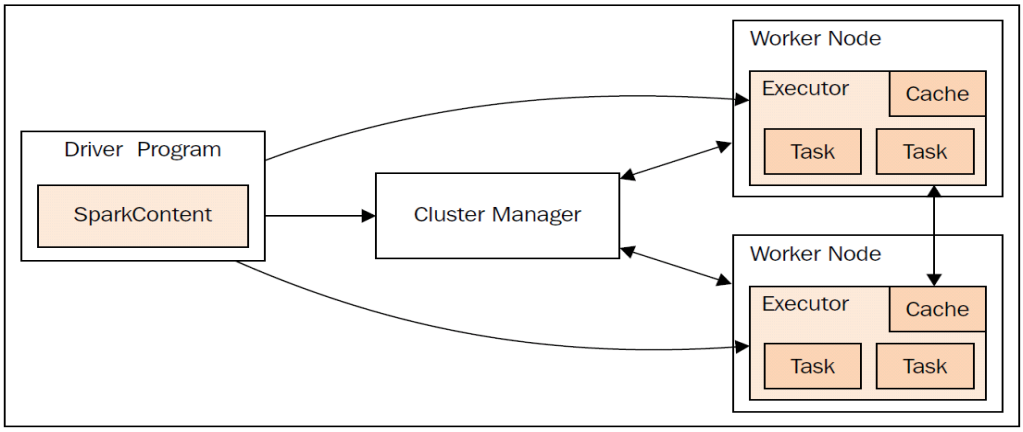



Figura 2.13: Gerenciamento de *clusters* do Spark (Frampton, 2015, p.8)

Há ainda a possibilidade de executar aplicações localmente em apenas uma máquina, neste caso é denominado modo *standalone*, sendo suportado o uso de *threads*, a fim de oferecer concorrência entre as aplicações, admitindo apenas o algoritmo FIFO como escalonador de processos.

Frampton (2015) lembra que o Spark não oferece armazenamento, assim depende de ferramentas de armazenamento de dados para leitura e escrita. Para isso pode ser usado o sistema de arquivos nativos do sistema operacional da máquina que executa o Spark, porém caso o conjunto de dados seja bastante elevado, pode se fazer necessário o uso de ferramentas de armazenamento de *big data*, como por exemplo, com o ambiente Hadoop, utilizando o HDFS.

Segundo Aven (2016), o Spark permite o desenvolvimento de rotinas de processamento de dados complexas, fornecendo uma API de alto nível e tolerante a falhas. Esse *framework* foi idealizado como alternativa ao uso do MapReduce do Hadoop, pois para aplicações que exigiam consultas interativa ou em tempo real com baixa latência o MapReduce demonstrou ser inadequado, devido à utilização do disco rígido para persistir dados intermediários durante o processamento dos dados.

Para implementar sua estrutura distribuída em memória e tolerante a falhas, o Spark utiliza o RDD e o dataframe, os quais preparam o conjunto de dados para serem distribuídos no *cluster*. O Spark procura maximizar o uso da memória dos *hosts* participantes do sistema distribuído, melhorando assim o desempenho da aplicação. A utilização do RDD ou dataframe em memória torna o Spark eficiente para operações interativas e para *machine learning*.

Por fim, conforme Aven (2016), o Spark suporta grande variedade de aplicações, as principais são:

- Operações de ETL;
- Análise preditiva e *machine learning*;
- Operações de acesso a dados com SQL;
- Mineração e processamento de textos;
- Processamento de eventos em tempo real;
- Aplicações gráficas;
- Reconhecimento de padrões; e
- Mecanismos de recomendação.

### 2.1.4 Scala

De acordo com Odersky (2016), A linguagem Scala agrega os paradigmas de programação: orientado a objetos e funcional. A combinação desses paradigmas torna possível expressar novos tipos de padrões de programação e abstração de componentes, levando a um estilo de programação mais legível e conciso.

Além disso, o Scala foi projetado para ser compatível com o Java, permitindo que seu compilador gere bytecodes para JVM. Seu desempenho é compatível com programas em Java, sendo possível chamar métodos, acessar campos, herdar classes e implementar interfaces em Java, sem a necessidade de adicionar nenhuma sintaxe extra, pois faz uso das bibliotecas do Java.

Segundo Duvvuri e Singhal (2016), a codificação de programas no Spark com linguagem Java é possível, pois utiliza a JVM, entretanto se comparada com o Scala, apresenta maior dificuldade de ser utilizada, necessitando de maior habilidade em Java.

Conforme Acodemy (2015), ao passo que se desenvolvem conhecimentos com o uso do Spark, aumenta-se a necessidade de configurar o Spark e utilizar suas API, para esse fim é exigido habilidades em Scala, em Python ou em ambos.

Assim é importante definir qual linguagem de programação aprender e aplicar em projetos com Spark. Por isso este autor comparou o Scala e o Python utlizando os seguintes critérios:

- Performance: quando não se utiliza apenas códigos em Python para solicitar bibliotecas Spark, o Scala chega a ter desempenho dez vezes superior ao Python;
- Curva de aprendizagem da linguagem: aqueles que não conhecem nenhuma dessas linguagens, o Python é mais fácil de aprender do que o Scala, além de possuir maior apoio de comunidades de desenvolvedores, livros e documentações.
- Facilidade de uso: como o Spark é construído usando o Scala, compreender o funcionamento do Spark se torna mais fácil quando já se conhece a linguagem Scala;
- Bibliotecas: o Python possui quantidade maior e melhor de bibliotecas de *machine learning* se comparado ao Spark, embora tenha menos bibliotecas, elas são direcionadas para *big data*. O Scala perde nesse quesito, pois há poucas ferramentas e bibliotecas para *data Science*, quando comparado ao Python.

Em relação à linguagem R, Duvvuri e Singhal (2016) definiram como sendo a mais adequada para análise de dados, porque possui um conjunto mais rico de bibliotecas para isso. Entretanto para análises em tempo real, o Scala é a melhor escolha por ser a linguagem utilizada pelo Spark. A partir da versão 1.4 do Spark, o R foi suportado para atrair cientistas que utilizam esta linguagem de programação.

Logo, caberá ao projetista avaliar as necessidades de seu projeto e identificar quais critérios citados são mais relevantes.

## 2.2 Metodologia

A metodologia adotada para o desenvolvimento deste projeto foi divida em cinco partes, conforme a figura 2.14.

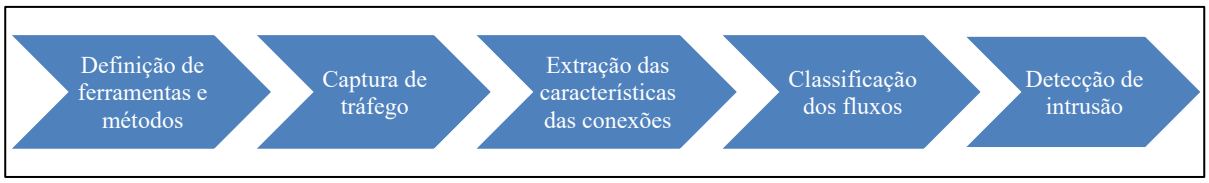



Figura 2.14: Etapas da metodologia proposta

### 2.2.1 Definição de ferramentas e métodos

Uma vez que há diversas ferramentas de *big data* disponíveis para auxiliar a resolução do problema proposto, bem como *softwares* para realizar a captura e análise de tráfego em redes de computadores, assim a tarefa de determinar as melhores ferramentas e técnicas a serem empregadas se torna difícil. Para isso, os critérios de seleção adotados foram:

- Ferramenta gratuita;
- Possibilidade de capturar e manipular pacotes por meio de *scripts*;
- Baixo custo computacional;
- Escalabilidade;
- Bibliotecas que facilitem a implementação do projeto; e
- Boa documentação de uso.

Após o estabelecimento desses critérios, foram pesquisadas em trabalhos relacionados ao tema quais ferramentas eram utilizadas e quais eram suas limitações.

### 2.2.2 Captura de tráfego

Para a execução da captura optou-se por utilizar o Tshark por que essa ferramenta permite a criação de *scripts* para realizar capturas personalizadas, a utilização de utilitários do Wireshark, a elaboração de relatórios sobre as características da captura e a manipulação dos arquivos PCAP. Ainda que sua execução não apresente paralelismo, o que pode limitar o desempenho do sistema.

Nessa parte do projeto, foram adotados alguns métodos de captura para tráfego normal e RAT. A fim de reduzir o tempo de execução da detecção, foi estudada a possibilidade de diminuir o tamanho da captura por meio da retirada de alguns campos de cabeçalhos do protocolos utilizados, os quais são desnecessários para análise das características dos ataques.

A captura do tráfego normal foi realizada em lotes de captura contínua em intervalos de cinco minutos, sendo que cada lote é avaliado individualmente pelo sistema de detecção. O tempo de captura do tráfego da rede interna estudada foi de aproximadamente uma semana ininterrupta. Essa captura foi utilizada para treinamento dos algoritmos de aprendizagem de máquina. E para testes do modelo foi usado uma captura de tráfego normal distinta com

tempo de captura de dez minutos do tráfego real, esse tempo de captura representa o dobro do requerido por que foi levado em consideração um aumento máximo da quantidade de pacotes de até 100% do tráfego da rede avaliada.

Agora, tendo em vista que o comportamento da conexão do RAT pode mudar de acordo com o modo de execução do RAT pelo atacante, pois dependendo do malware, diversas ferramentas podem ser utilizadas para analisar o computador da vítima. Logo, existiu a necessidade de adotar uma metodologia para realizar a captura do tráfego dos RAT, assim, após o estabelecimento da conexão, foram definidos três tipos de ações padrão do atacante:

- Apenas manter a conexão, sem executar nenhum comando;
- Executar diversos comandos que não geram grande quantidade de tráfego, tais como listar diretório, processos, registros e configurações do sistema operacional; e
- Executar o comando de capturar a tela da vítima em intervalos de tempos constantes, a fim de gerar grande tráfego.

Consequentemente, foram geradas três capturas para cada tipo de RAT analisado com tempo de execução da captura de cinco minutos.

Para realizar essas capturas, foi montado um laboratório com três máquinas virtuais (VM), conforme configurações expostas na tabela 2.1, O software utilizado para criar e gerenciar as VM foi o VirtualBox em sua versão 5.1.28.

Tabela 2.1: Configuração das máquinas virtuais para captura

| Sistema Operacional | Memória RAM | Processador | HD | Interface de rede |
|---|---|---|---|---|
| Windows XP 32-bit | 512 MB DDR3 1333 Mhz | 1x Intel i5-4570 3.20Ghz | 10 GB | 1 |
| Lubuntu 64-bit | 1024 MB DDR3 1333 Mhz | 1x Intel i5-4570 3.20Ghz | 24 GB | 2 |

Após a criação e configuração das VM, foi necessário conectá-las para efetuar a comunicação entre a máquina vítima e atacante, para isso elaborou-se uma topologia de rede que utiliza a máquina com sistema operacional Lubuntu para interligar o tráfego entres as duas máquinas com Windows XP, de acordo com a figura 2.15.

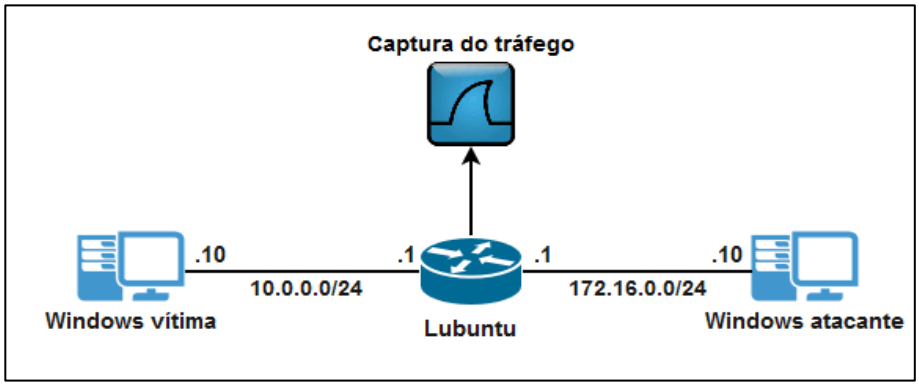



Figura 2.15: Topologia para captura de tráfego RAT

As máquinas foram separadas em dois segmentos de rede: 10.0.0.0/24 e 172.16.0.0/24, tendo como *gateway* as duas interfaces de rede do Lubuntu que foram configuradas com endereço IP 10.0.0.1 e 172.16.0.1.

Em seguida, a interface de rede da máquina Windows vítima foi configurada com o IP 10.0.0.10 e o Windows atacante com o IP 172.16.0.10.

Após testes e confirmação de conectividade bem sucedida, foi instalado o Tshark no Lubuntu para efetuar a captura na interface ligada ao Windows vítima, assim o ambiente ficou pronto para realizar a captura do tráfego TCP dos RAT.

### 2.2.3 Extração das características das conexões

Para realizar a extração das características de conexões dos arquivos de captura foi utilizado o Tshark, por meio do comando "-z conv", o qual gera um relatório predefinido da comunicação dos computadores em formato TXT. Outra forma explorada de extrair as características foi a utilização dos comandos "-T fields" e "-T json", que possibilita exportar os campos das capturas para os formatos CSV e JSON, respectivamente. Assim a opção de exportar os campos oferece uma oportunidade de complementar as características que não foram definidas no relatório do comando "-z conv".

Porém ao implementar o cálculo de algumas características, obteve-se tempo de execução extremamente alto, pois com as capturas de testes foram necessários, aproximadamente, 50 minutos para terminar esse tratamento de características. Logo tornou-se inviável essa complementação com o código empregado, por causa da restrição de execução do projeto que é de cinco minutos.

Ainda nessa parte de desenvolvimento do projeto, foi avaliada a evolução do tamanho do arquivo exportado em CSV e JSON, com intuito de auxiliar a escolha do formato que oferece melhor desempenho.

### 2.2.4 Classificação dos fluxos

A partir da definição do formato CSV como entrada para o *framework* Apache Spark que foi escolhido devido ao desempenho, escalabilidade e bibliotecas que auxiliaram o desenvolvimento do projeto, foi utilizado o módulo SQL do Spark para efetuar leitura dos arquivos com as características extraídas, a normalização dos valores e o cálculo das características complementares.

A implementação dos algoritmos de aprendizagem de máquina K-*means* e *Random Forest* foi possível com o uso do módulo MLlib do Spark. A utilização do K-*means* ofereceu novas características para as conexões a partir da definição de grupos similares, o que auxiliou a classificação dos fluxos com o algoritmo *Random Forest*, sendo classificados em normal, quando a coluna definida como "label" tinha valor igual a zero, e RAT quando seu valor era igual a um.

### 2.2.5 Detecção de intrusão

A última parte da metodologia tem como objetivo validar o modelo proposto, a fim de verificar a eficácia do sistema de detecção. Para isso, foi utilizada a

*confusion matrix* e as métricas de desempenho *accuracy*, *precision*, *recall* e F *measure* em dois estudos de casos, sendo um com capturas de RAT seguindo o comportamento da conexão do RAT definido na parte da metodologia, denominada captura de tráfego, e outro estudo de caso com capturas de terceiros.

## 2.3 Trabalhos relacionados

Esta seção apresenta o estado da arte sobre detecção de RAT, pois relata os principais artigos publicados acerca do problema, os quais definem as características dos RAT e métodos de detecção. Além de comentar os resultados dos trabalhos, foram adicionadas partes dos trabalhos para melhorar a compreensão, porque se trata de tema não consolidado na literatura.

### 2.3.1 *Intrusion Detection & Prevention Approaches*

Bijone (2016) elaborou uma pesquisa com o objetivo de apresentar o estado da arte dos métodos de detecção de intrusão e prevenção, bem como seus desafios e limitações. O autor considerou estudos com aplicações de *machine learning* na segurança de redes, e, neste caso, apresentou o método de validação de desempenho de um sistema de detecção de intrusão, explicando detalhadamente alguns casos que podem influenciar nos resultados.

O método descrito é o *confusion matrix*, tabela 2.2, que avalia a saída de um sistema de detecção, o qual poderá ter apenas quatro saídas. No entanto esse critério de avaliação, quando utiliza *machine learning,* é mais adequado em algoritmos supervisionados classificadores, como por exemplo, o algoritmo *Random Forest.*

Tabela 2.2: *Confusion Matrix* (Bijone, 2016)

| | **Normal** | **Attack** |
|---|---|---|
| **Normal** | True Negative (TN) | False Positive (FP) |
| **Attack** | False Negative (FN) | True Positive (TP) |

Bijone (2016) descreveu:

- *True negatives* (TN) – eventos normais que foram classificados como normais;
- *True positives* (TP) – eventos anormais (ataques) que foram classificados como anormais;
- *False positives* (FP) - eventos normais que foram classificados como anormais; e
- *False negatives* (FN) - eventos anormais que foram classificados como normais;

Após essa classificação, pode-se calcular a taxa de ocorrência de cada caso, e assim verificar a eficiência do sistema:

- *False Positive Rate* (FPR) = FP / (FP +TN);
- *False Negative Rate* (FNR) = FN / (FN+TP);
- *True Positive Rate* (TPR) = TP / (TP+FN);

- *True Negative Rate* (TNR) = TN / (TN+FP);
- *Accuracy* = (TP+TN) / (TP+TN+FN+FP); e
- *Precision* = TP / (TP+FP).

O autor afirma que para se ter um IDS efetivo, tanto FP quanto FN deverão ser reduzidos, enquanto *Accuracy*, TP e TN deverão ser maximizadas.

Além dessas métricas de desempenho, Ruiter (2015) apresenta outras formas de medidas aplicadas em modelos de aprendizagem que fornecem saídas binárias (positiva ou negativa), e seus erros (falsos positivos e falsos negativos).

Essa mensuração combina as saídas mais relevantes: true positives (TP), false positives (FP) e false negatives (FN), e exibe a taxa para avaliar o desempenho do modelo. A seguir será apresentado o cálculo, segundo Ruiter (2015), do recall e F measure:

- *Recall* (taxa de correta previsão de positivos) – R = TP / (TP+FN); e
- *F measure* (média harmônica entre precision e recall) = 2*(P*R) / (P+R).

### 2.3.2 *Remote Administrative Trojan/Tool* (RAT)

Kondalwar e Shelke (2014) procuraram, em seu trabalho, descrever as características desse tipo de *malware*. RAT pode ser compreendido como software de acesso remoto que permite o controle do computador ao ser instalado um servidor na máquina da vítima, possibilitando a comunicação com o atacante, por meio de portas dos protolocos *Transmission Control Protocol* (TCP) e *User Datagram Protocol* (UDP).

Segundo os autores, os meios de infecção mais comuns são: anexos de e-mails, técnicas de *spamming*, *softwares* de mensageria (Yahoo *Messenger* e Skype) e partes de códigos de programas aparentemente legítimos.

Kondalwar e Shelke (2014) classificaram as conexões em dois tipos: conexão direta e reversa. Na conexão direta, o RAT possui configuração simples e os clientes podem se conectar a um ou mais servidores diretamente. Neste caso, a conexão é iniciada pelo atacante, e pode ser interrompida pelos ativos de segurança da corporação.

Já, na conexão reversa, a máquina infectada inicia a conexão com o atacante, ou seja, o servidor inicia a comunicação, assim os dispositivos de segurança não podem impedir a conexão. Basicamente, nessa arquitetura a máquina infectada se comporta como o servidor e o atacante como o cliente.

Conforme esses autores, os RAT podem capturar telas e teclas digitadas, obter informações de contas de usuários, senhas e informações sensíveis. A partir de suas observações, os autores identificaram as principais ações desse *malware*:

- *Download, upload,* apagar, e renomear de arquivos;
- Formatar unidades de disco;
- Ejetar CD-ROM;
- Disseminar vírus e *worms*;
- Capturar teclas do teclado;

- Obter senhas e números de cartões bancários;
- Visualizar, terminar e iniciar processos de sistema;
- Exibir textos e reproduzir músicas; e
- Mover ou clicar o ponteiro do *mouse*.

Portanto, o RAT fornece possibilidades de comandos para gerenciar arquivos, agendamento de processos e configuração do sistema operacional da vítima. Para ilustrar isso, Kondalwar e Shelke (2014) mostraram as principais ferramentas administrativas do sistema operacional Windows utilizadas por RAT em seu funcionamento:

- MSRPC - Win32 *legacy management Application Programming Interface* (API) – implementação do *Distributed Computing Environment* (DCE) / *Remote Procedure Call* (RPC), que utiliza o SMB;
- *Windows Management Instrumentation* (WMI) - ferramenta de monitoramento de eventos do sistema Windows;
- Ferramentas de acesso remoto gráfico, por exemplo, o *Remote Desktop Protocol* (RDP);
- Comandos via *Command Line Interface* (CLI) – utilizando o PsExec (similar ao telnet); e
- Ferramentas *web*

Apesar de o estudo elaborado por Kondalwar e Shelke (2014) levar em consideração apenas a plataforma Windows, por ser o principal alvo de ataques, o RAT pode ser aplicado a outras plataformas, observando-se a peculiaridade de cada sistema.

Outra característica importante do RAT é a capacidade de sequestro de conexões de rede, a qual permite a interceptação de dados ou a inserção de mensagens falsas. Há também a opção de ação de captura de pacotes, o que facilita monitorar as atividades das vítimas, bem como identificar a topologia do alvo.

Além disso, foi relatado um procedimento adotado geralmente para ativar o *malware*, bem como algumas configurações possíveis. Antes de instalar o RAT, é possível customizar a forma de funcionamento, definindo a porta TCP ou UDP utilizada, os métodos de gatilhos de iniciação, os algoritmos de encriptação e o usuário/senha para acessar a aplicação.

Durante a instalação do RAT, esse pode ser associado a programas legítimos, tais como o "IEXPLORE.EXE". Assim, o servidor poderá ser executado toda vez que a aplicação iniciar, e rodará em *background* o seu serviço, o qual é escutar em uma porta determinada, à espera de um comando do atacante.

No apêndice A é possível visualizar um exemplo de procedimento de instalação e utilização do RAT Poison Ivy versão 2.3.2.

### 2.3.3 *A General Framework of Trojan Communication*

Conforme Li et al. (2012), a detecção de *trojan* é dividida em duas categorias: *host-based* e *network-based*. A primeira tem o intuito de observar o comportamento do *trojan* na máquina da vítima, usando as chamadas de sistemas e API dos sistemas operacionais como

parâmetros de detecção. Enquanto a segunda, analisa o tráfego de todas as máquinas de uma determinada rede de computadores, monitorando os pacotes enviados e recebidos.

Como os *trojans* possuem técnicas de metamorfismo (capacidade de alterar o próprio código), as chamadas de sistema podem ser facilmente modificadas e novas chamadas irrelevantes podem ser adicionadas. Somado a isso, tem-se a obrigação de entender todas as chamadas de sistemas de diferentes sistemas operacionais, havendo também a necessidade de se instalar *softwares* de monitoramento em todos os *hosts* que pertencem à rede de computadores analisada, ocasionando assim maior uso de recursos. Dessa forma, torna-se muito difícil padronizar o comportamento na categoria *host-based*.

Diferentemente, na categoria *network-based*, não há necessidade de se instalar *softwares* nesses hosts, pois pode utilizar abordagens de análise de pacotes, tais como a DPI, que analisa todos os campos do pacote, ao capturar o tráfego em ativos de rede. Atualmente, esta categoria de detecção procura identificar assinaturas de *malwares* conhecidos, possuindo as seguintes limitações: ineficiência para detectar novos tipos de *trojan* sem assinaturas; remodelagem do método de detecção à medida que se altera a forma de ação do *trojan*; aumento significativo da base de assinaturas disponíveis, devido à grande quantidade de malwares criados diariamente, exigindo, portanto alto custo computacional para identificar a ameaça; e por fim, quando se usa criptografia no *payload* (carga útil), dificulta-se a verificação de todo conteúdo trafegado.

Esses autores analisaram trabalhos anteriores a fim de identificar as características de conexões necessárias para classificar o fluxo de um *trojan*, e definiram os seguintes parâmetros como suficientes para classificar as conexões:

- Duração do fluxo de pacotes;
- Intervalo de tempo entre pacotes;
- Total de pacotes enviados e recebidos;
- Total de dados enviados e recebidos; e
- Duração da sessão de comunicação.

A partir desses parâmetros é possível obter outros que facilitarão a distinção entre o comportamento normal de aplicações da rede e o comportamento dos *trojan*.

Além disso, Li et al. (2012) verificaram os conceitos de conexões principais e de conexões secundárias, sendo aquelas utilizadas para estabelecer conexão, enviar *keep-alive* e comandos, enquanto estas utilizadas para roubar informações. Por isso acrescentaram outras características, que são:

- *Out-in-pkts* – verifica se o total de segmentos enviados do servidor para o cliente é superior ao do cliente para o servidor. Se for maior, o valor da característica é 1, caso contrário será 0;
- *Out-in-bytes* – mesma verificação do *out-in-pkts*, porém em vez de analisar o total de segmentos, é observado o total de *bytes*;
- *Duration-versus* – compara a duração da sessão do endereço de origem e destino, com a duração da conexão principal; e
- Mean interval – média do intervalo de tempo de envio dos segmentos da conexão principal.

Os autores desenvolveram um modelo de detecção *network-based* por anomalias, denominado Manto, figura 2.16, cujo objetivo é determinar dois tipos de perfis de comportamento: um para os *trojan* e outro para as aplicações normais. Este *framework* consiste em três etapas:

- Cálculo das características das conexões;
- Construção dos perfis com o algoritmo de *machine learning*; e
- Detecção do *trojan*.

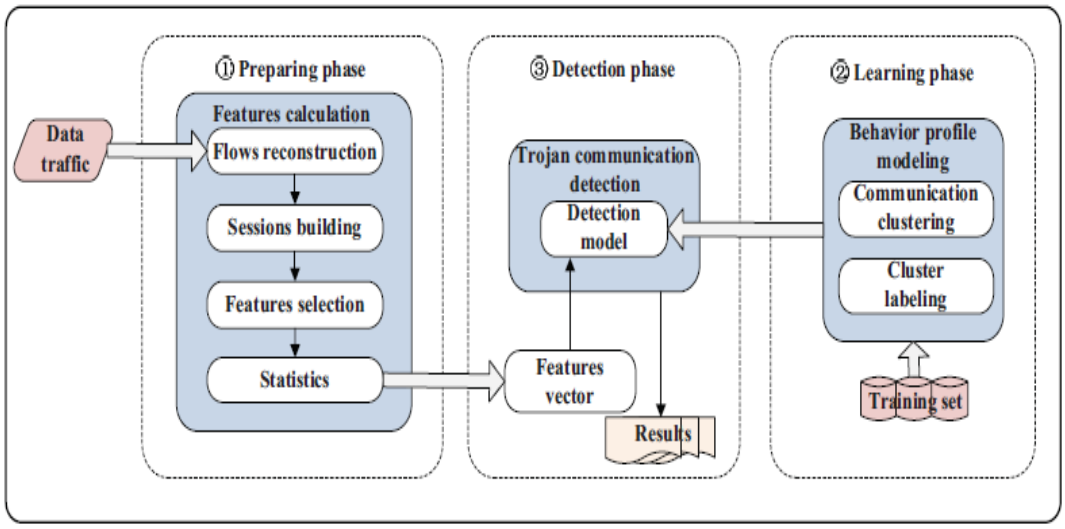



Figura 2.16: Arquitetura Manto (Li et al., 2012)

Durante a implementação da etapa de cálculo das características, verificaram-se os parâmetros citados, considerando somente a comunicação que utiliza o protocolo TCP. Devido ao fato de a conexão ser orientada, é possível identificar o início do estabelecimento da conexão, ao examinar o TCP 3-*way handshake*, e coletar as características de conexão.

Embora o término da uma conexão seja efetuada com o recebimento de segmentos com *flag* FIN ou RST setada, os autores estabeleceram que o fim de uma conexão seria após cinco minutos, caso não fossem enviados segmentos com essas *flags*. Sem esse limite, talvez fosse difícil detectar rapidamente a ação do trojan, pois a conexão pode facilmente ultrapassar esse período.

Em seguida, eles agruparam os segmentos das conexões estabelecidas em pares de endereços de origem e destino, reconstruindo as sessões de comunicações criadas, , e, dessa forma, obtiveram todos os parâmetros necessários para gerar as estatísticas das conexões.

Antes de prosseguir para a próxima etapa, foi necessário ajustar a escala de valores das características utilizadas, senão uma teria maior peso do que a outra. Para isso Li et al. (2012) utilizaram a equação de transformação (1), o qual $A_i$ é o valor original da característica, $A_{min}$ é o valor mínimo, $A_{max}$ é o valor máximo, e o $A'_i$ é o novo valor redimensionado.

$$A'_i = \frac{A_i - A_{min}}{A_{max} - A_{min}} \qquad \text{Eq. (1)}$$

Na etapa posterior, denominada construção dos perfis, os autores classificaram as sessões em normal e anormal. Primeiro utilizaram algoritmo de *clustering* que agrupou as

sessões com características similares, após isso, cada grupo recebeu um das duas classificações. O algoritmo utilizado foi o K-*Means* que é um algoritmo de *clustering* em aprendizagem não supervisionada de máquina, cuja aplicação foi comparada com outros algoritmos similares no trabalho de Erman et al. (2006).

Baseado neste estudo, Li et al. (2012) escolheram esse algoritmo para sua implementação, porque os resultados mostraram que esse algoritmo é eficiente para classificação de fluxos de tráfego e pode oferecer agrupamento para cada tipo de aplicação. Além disso, eles o definiram como um algoritmo simples e conveniente de se implementar, uma vez que que outros algoritmos de *clustering* necessitam de maior tempo de aprendizagem do que o K-Means.

Para classificar a conexão, foi utilizado o conceito de *maximum likelihood estimation*, que é um modelo probabilístico para determinar semelhanças entre as amostras, em relação aos grupos resultantes da aplicação do algoritmo de *clustering*. De acordo com o resultado, foram encontradas três possibilidades: normal, anormal e desconhecido. As conexões que os autores não conseguiram identificar foram enquadradas no último caso e serão analisadas em trabalhos futuros.

Li et al. (2012) observaram, ao analisar as capturas que, sob a perspectiva da máquina da rede interna, 80% das conexões tinham mais pacotes recebidos do que enviados, e 60% tinham mais *bytes* recebidos do que enviados. Assim concluíram que normalmente o tráfego de entrada na rede é maior do que o de saída, que os *trojans* possuem o comportamento inverso e que as conexões com o protocolo *Peer-to-Peer* (P2P) podem apresentar comportamento similar aos *trojans,* porque fazem *download* e *upload* de arquivos.

Além disso, eles notaram que 72% dos pacotes possuíam intervalos de envio inferior a um segundo, e 94% apresentaram esse intervalo inferior a cinco segundos. Como há atrasos em uma rede interna, o que pode influenciar esse cálculo, os autores adotaram como parâmetro a média desse intervalo.

As métricas utilizadas para avaliar a detecção da arquitetura deles foram acurácia, que é a razão entre o total de *trojan* identificados pelo total testados, e a taxa de falso positivo, que é a porcentagem de conexões incorretamente identificadas como *trojan*.

O modelo apresentou até 91% de acurácia e 3.2% de falso positivo, entretanto o sistema precisou previamente de todo o comportamento dos *trojans* para identificar corretamente o *malware*, o que geralmente não é obtido em processos em tempo real, pois apenas uma parte é obtida.

### 2.3.4 *An Unknown Trojan Detection Method*

Yu et al. (2013) apresentaram outro modelo de detecção de *trojan,* utilizaram como parâmetros de detecção as características de fluxos de conexão:

- Número de conexões – eles identificaram que aplicações normais utilizam geralmente múltiplas conexões, enquanto os *trojans* apenas uma ou duas;
- Número de conexões de *upload* - Aplicações normais utilizam muitas conexões, e os *trojans* poucas;
- Intervalo de tempo entre as conexões concorrentes – Aplicação normal possui um intervalo curto, no entanto os *trojans* gastam um intervalo longo;

- Número de endereços IP distintos – Aplicação normal usa muitos endereços, e *trojans* poucos;
- Taxa de tráfego de saída e entrada – nas conexões de *trojan*, a quantidade de dados enviados é superior à quantidade recebida, considerando a conexão reversa.

A categoria de detecção escolhida pelos autores foi *host-based*, a qual possui as desvantagens citadas anteriormente. A figura 2.17 ilustra a arquitetura idealizada pelos autores.

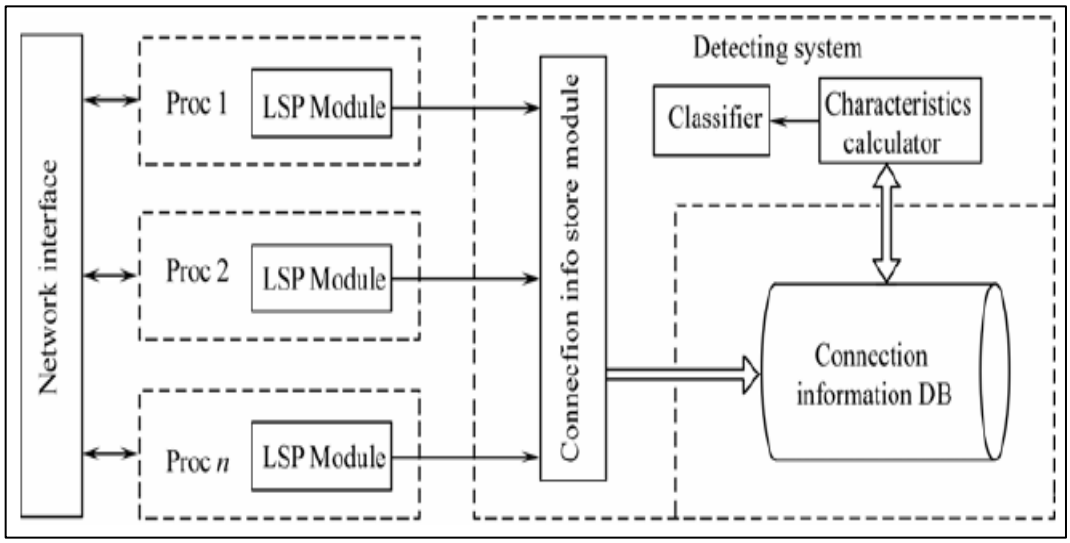



Figura 2.17: Sistema de detecção de *trojan host-based* (Yu et al., 2013)

Esse modelo apresenta a mesma quantidade de etapas do modelo anterior, as quais são: coleta de dados, cálculo das características das conexões, classificação dos fluxos e resultados de detecção. Contudo, este estudo combinou os algoritmos de aprendizagem supervisionada *decision tree e* Naïve Bayes para classificar as conexões.

Porém, não ficou claro como a escolha das características influenciaram no resultado, bem como os seus valores. O que foi possível compreender foi que a característica de número de endereços IP distintos representou maior peso na detecção, pois foi apresentada como raiz do modelo de aprendizagem utilizado, e de acordo com esse valor, o fluxo era classificado em uma folha da *decision tree,* conforme figura 2.18, o qual NB1 e NB2 são a classificação do Naïve Bayes.

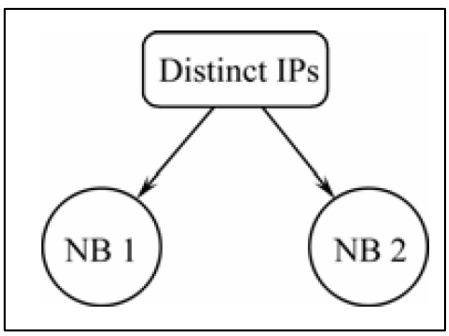



Figura 2.18: Modelo treinado com *decision tree* e Naïve Bayes (Yu et al., 2013)

Embora não tenha sido um estudo muito claro, apresentou outra forma de se detectar *trojans*, e contribuiu com a combinação de dois algoritmos de *machine learning*, com o intuito de fornecer melhor acurácia, diferentemente do modelo anterior que utilizou apenas um.

### 2.3.5 *An Approach to Detect Remote Access Trojan*

Com o intuito de identificar atividades de RAT, Jiang e Omote (2015) elaboraram um estudo para detectar essa invasão em estágio inicial de comunicação. A vantagem desse modelo é a possibilidade de detectar o RAT antes de o atacante fazer a análise da vítima ou realizar o roubo de informações.

Segundo eles, para identificar um RAT, em estágio inicial de ataque, é necessário observar o TCP 3-*way handshake* do RAT. Considerando o caso de conexão reversa, tem-se o servidor (*host* infectado) iniciando a conexão ao enviar um segmento com a *flag* SYN para o cliente (atacante); em seguida, o atacante responde com um segmento contendo a *flag* SYN/ACK; e por fim o servidor responde com um segmento que há a *flag* ACK, assim a conexão é estabelecida.

Foi verificado que o tempo de estabelecimento de conexão do RAT é bem inferior ao tempo de aplicações normais, conforme figura 2.19, e que isso ocorre devido à necessidade de se esconder o tráfego gerado, para que não seja percebido por alguma ferramenta de segurança.

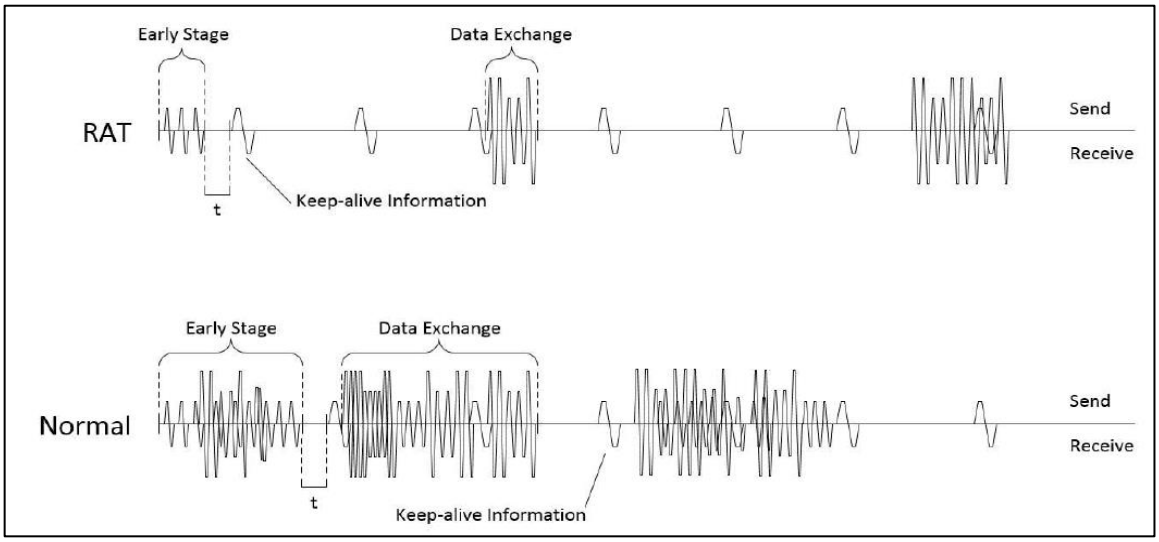



Figura 2.19: Comportamento inicial da conexão de RAT (Jiang e Omote, 2015)

Jiang e Omote (2015), ao analisarem o tempo total de sessão de um RAT, identificaram que normalmente termina antes de 100 pacotes. Portanto as características do tempo de conexão e intervalo de pacotes se mostram importantes parâmetros para classificação.

O modelo de detecção proposto por esses autores possui três etapas: extração das características, treinamento do algoritmo de *machine learning* e identificação da ameaça.

A abordagem de classificação escolhida foi de apenas rotular o fluxo em normal ou anormal. Para isso os autores analisaram o desempenho de cinco algoritmos supervionados de *machine learning*.

Assim, Jiang e Omote (2015) apresentaram outros tipos de algoritmos de aprendizagem de máquina, e fizeram um estudo de desempenho entre eles ao aplicá-los na detecção de RATs em estágio inicial de comunicação. Segundo esses autores, os algoritmos e suas definições são:

- *Decision Tree* (DT) – estrutura de árvore que se divide em ramos, escolhendo a melhor opção de classificação de acordo com as características dos dados de entrada;
- *Support Vector Machine* (SVM) – necessita de separação em fronteiras de decisão, para permitir que os dados em treinamento tenham um limite (margem) máximo de decisão. Assim, para que se tenha maior desempenho, deve-se aumentar essa margem ao máximo;
- *Naive Bayes* (NB) – é definido como um algoritmo simples e eficiente que processa grande quantidade de dados rapidamente, utilizando a equação de probabilidade condicional de Bayes para escolha da classe;
- *K-Nearest Neighbor* (KNN) – a classificação dos dados é decidida a partir do cálculo de vizinhos, a fórmula utilizada é a da distância euclidiana;
- *Random Forest* (RF) – é um conjunto de *Decision Trees*, as quais são geradas randomicamente. Comparam-se os resultados dessas árvores para que os melhores sejam escolhidos. Esse algoritmo proporciona maior estabilidade e acurácia do que apenas um *Decision Tree*.

As características de conexão adotadas pelos autores podem ser vistas na tabela 2.3. Foram extraídos dos fluxos capturados os seguintes parâmetros: número de pacotes, tamanho de dados enviados, tamanho de dados recebidos, número de pacotes enviados, número de pacotes recebidos, taxa de número de pacotes enviados/recebidos e taxa de tamanho de dados enviados/recebidos.

Tabela 2.3: Características de conexão em estágio inicial (Jiang e Omote, 2015)

| Feature | Description |
|---|---|
| PacNum | packet number |
| OutByte | outbound data size |
| OutPac | outbound packet number |
| InByte | inbound data size |
| InPac | inbound packet number |
| O/Ipac | rate of OutPac/InPac |
| OB/OP | rate of OutByte/OutPac |

Jiang e Omote (2015) resumiram graficamente o processo de detecção elaborado por eles, conforme figura 2.20. Percebe-se que os algoritmos necessitam de classificação prévia dos fluxos para sejam identificados novos fluxos.

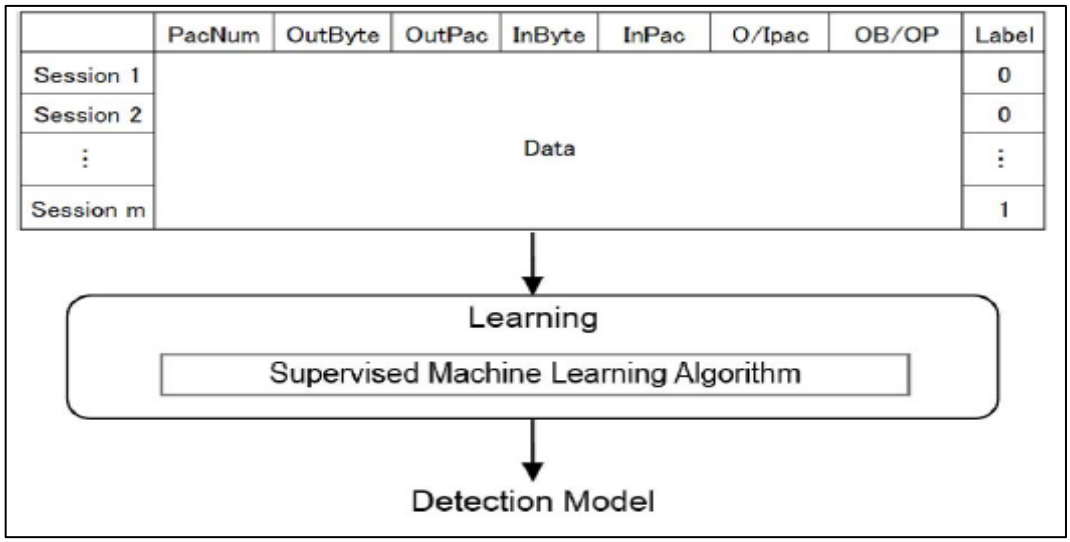



Figura 2.20: Detecção com algoritmo supervisionado (Jiang e Omote, 2015)

O resultado do teste de desempenho dos algoritmos de *machine learning* testado está sumarizado na tabela 2.4, a qual FPR significa taxa de falso positivo e FNR taxa de falso negativo. Essa tabela mostra que o *Random Forest* apresenta melhor performance para esse tipo de aplicação.

Tabela 2.4: Resultado do comparativo dos algoritmos (Jiang e Omote, 2015)

| Item | Accuracy | FPR | FNR |
|---|---|---|---|
| RF | 0.971 | 0.023 | 0.100 |
| DT | 0.960 | 0.030 | 0.200 |
| SVM | 0.533 | 0.458 | 0.600 |
| NB | 0.430 | 0.597 | 0.100 |
| KNN | 0.919 | 0.054 | 0.500 |

A fim de comparar o impacto das características selecionadas, os autores fizeram um estudo de combinação das características. Para o caso do *Random Forest,* nota-se que seu desempenho é aprimorado à medida que se adiciona características ao modelo, conforme a figura 2.21.

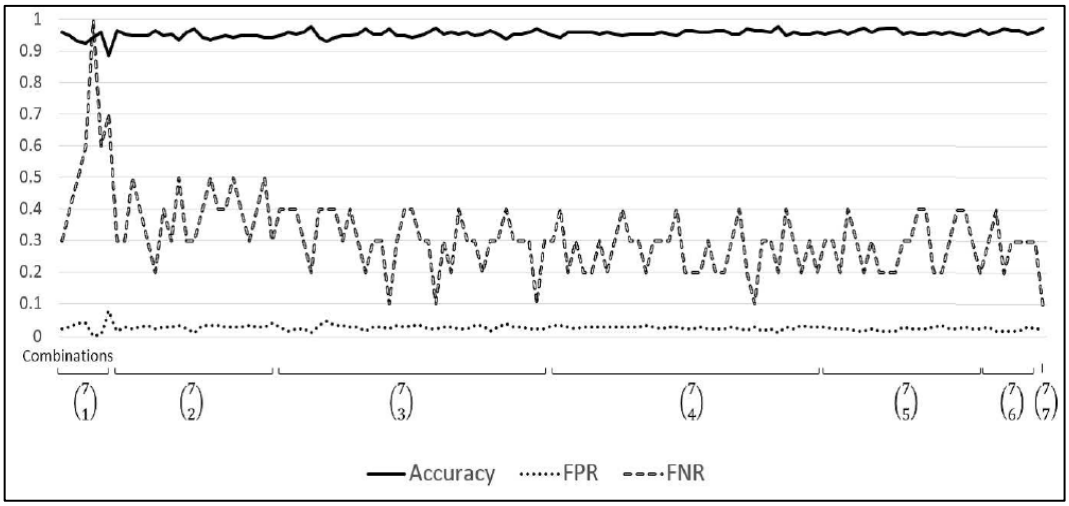



Figura 2.21: Combinação de características com *Random Forest* (Jiang e Omote, 2015)

Eles ainda compararam a frequência de cada característica na captura efetuada. Na tabela 2.5, pode-se visualizar o comportamento dos parâmetros para aplicações normais e RAT.

Tabela 2.5: Comportamento das características analisadas (Jiang e Omote, 2015)

| Feature | Type | Trend |
|---|---|---|
| PacNum | N | 78% are more than**10pack** |
| | R | 20% are more than**10pack** |
| OutByte | N | 93% are more than **500byte** |
| | R | 20% are more than **500byte** |
| OutPac | N | 52% are more than **10pack** |
| | R | 10% are more than **10pack** |
| InByte | N | 71% are more than **500byte** |
| | R | 10% are more than **500byte** |
| InPac | N | 38% are more than **10pack** |
| | R | 10% are more than **10pack** |
| O/Ipac | N | 99% are more than **1** |
| | R | 60% are more than **1** |
| OB/OP | N | 68% are more than **100byte/pack** |
| | R | 30% are more than **100byte/pack** |
| * N: Normal Application , R: RAT | | |

Por fim, o trabalho elaborado por Jiang e Omote (2015) apresentou qualidade metodológica e clareza nos parâmetros adotados, bem como contribuiu bastante com um novo método de detecção; Entretanto foram testados apenas dez RAT diferentes e, assim como os trabalhos mostrados anteriormente, não ficou claro o método de testes dos RAT. O comportamento do RAT é dinâmico e pode não somente manter a conexão, como também enviar comando de tempos em tempos e, até mesmo, gravar a tela da vítima, sendo que este último justifica o maior tráfego de saída. Dessa forma, faltou explicitar o comportamento definido para análise.

### 2.3.6 *RAT-based Malicious Activities Detection*

Yamada et al. (2015) destacaram em sua obra o ataque que utiliza o RAT como meio de roubar informações de máquinas não infectadas que estão na mesma rede interna. Isso é possível devido à combinação de outra técnica, *Pass-the-hash*, que rouba as credenciais da vítima para usar os protocolos remotos, tais como o SMB, DCE/RPC e RDP, e aproveita suas vulnerabilidades para executar o acesso indevido. O uso dessa técnica é ilustrado na figura 2.22.

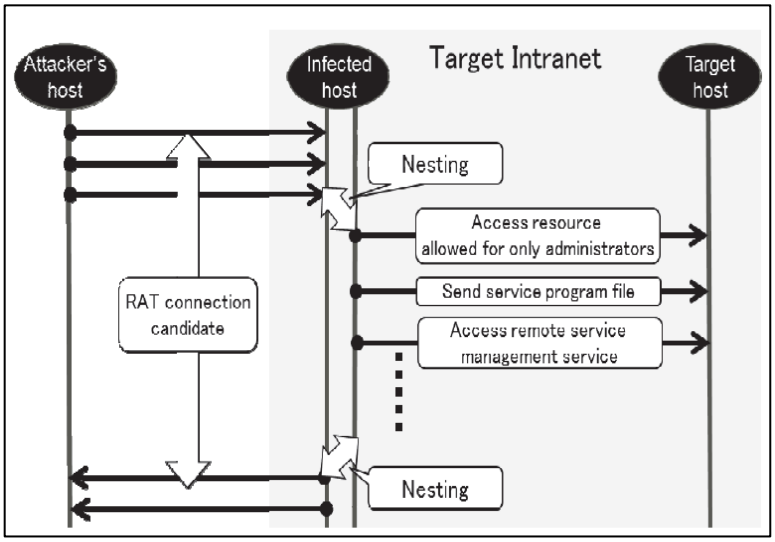



Figura 2.22: Exploração de ferramentas de acesso remoto (Yamada et al., 2015)

Eles analisaram 43 tipos de RAT, todos funcionando com conexão reversa. Logo o servidor é instalado na máquina infectada e o cliente no atacante. Neste caso, a conexão é iniciada pelo servidor. Na tabela 2.6, é possível observar as funcionalidades dos RAT analisados

Tabela 2.6: Funcionalidades de 43 RAT diferentes (Yamada et al., 2015)

| Function | Type | Percentage |
|---|---|---|
| Kill process | environment management | 97.67 % (42/43) |
| Start/stop service | environment management | 76.74 % (33/43) |
| Edit registry | environment management | 69.77 % (30/43) |
| List active window | environment management | 88.37 % (38/43) |
| Search/download file | environment management | 86.05 % (37/43) |
| Capture screen | environment management | 83.72 % (36/43) |
| Execute program | cooperation with other tool | 100.0 % (43/43) |
| Command prompt | cooperation with other tool | 79.07 % (34/43) |
| Upload file | cooperation with other tool | 86.05 % (37/43) |
| Uninstall server | server management | 93.02 % (40/43) |
| Update server | server management | 60.47 % (26/43) |

Os autores classificaram os tipos de funções dos comandos em:

- *Environment management* – comandos que promovem mudanças do ambiente e roubo de informações;

- *Cooperation with other tool* – comandos para executar outros programas que não fazem parte do RAT; e
- *Server management* – comandos para atualizar ou apagar o servidor RAT.

Durante os testes, assumiram que os RAT possuíam comportamento ativo e interativo, ou seja, estabeleciam conexões e capturavam dados da máquina infectada. Yamada et al. (2015) conseguiram identificar outras características da conexão de um RAT, tabela 2.7, além das identificadas em trabalhos realizados anteriormente.

Tabela 2.7: Novas características identificadas (Yamada et al., 2015)

| Number | Feature of RAT communication | Percentage |
|---|---|---|
| 1 | Connect form server to client | 100.0 % (43/43) |
| 2 | Encrypted communication | 53.49 % (23/43) |
| 3 | Push communication | 100.0 % (43/43) |
| 4 | Sustain remote control connection | 100.0 % (43/43) |
| 5 | Establish sub-connection | 41.86 % (18/43) |

A primeira característica da tabela 2.7, é resultado da escolha por RAT apenas com conexões reversas. Já a terceira característica, utilização da *flag* PUSH do TCP, é uma característica interessante para se usar no sistema de detecção, pois está presente em todas as conexões do RAT.

Segundo Jiang e Omote (2015), além dos RAT, os serviços que geralmente possuem grande quantidade de segmentos com essa *flag* são comunicações P2P e *cloud*. Embora seja difícil distinguir apenas com essa característica, poderá ajudar a reduzir para um grupo menor de conexões que compartilham essa característica.

Outra característica presente em todas as conexões de RAT é a capacidade de manter a conexão, enviando *keep-alive* e transmitindo comandos de tempos em tempos para não ser detectado. E em relação às demais características, acabaram não representando bem o comportamento de um RAT, pois aparecem em apenas na metade dos casos.

O sistema de detecção implementado por Yamada et al. (2015) utilizou as seguintes características:

- Verificou se a porta TCP de destino está permitida (porta do servidor RAT);
- Verificou se a conexão HTTP *proxy* usa método CONNECT;
- Verificou se os segmentos possuem *flag* PUSH; e
- Verificou o tempo de das conexões.

Eles monitoraram 50 a 60 milhões de conexões TCP, e verificaram que 0.0159% mantiveram conexões até dez segundos, enquanto apenas 0.0005% sustentaram acima de quatro minutos. Essa é uma característica que pode ser usada para identificar RAT em rede interna.

Apesar de o estudo apresentar contribuições e ser aplicado em conjunto com protocolos de acesso remoto legítimos, faltou demonstrar a arquitetura implementada e os

algoritmos, uma vez que os autores apenas ilustraram os procedimentos e os resultados, apontando de forma superficial os aspectos metodológicos.

### 2.3.7 *Big Data to Detect Remote Access Trojans*

Uma abordagem mais recente foi desenvolvida por Pallaprolu et al. (2016), a qual utiliza ferramentas de *big data* para implementar um *framework* com intuito de classificar os fluxos de dados. O objetivo principal desse trabalho foi o de encontrar um método de detecção capaz de classificar um grande fluxo a partir de pequena amostra de fluxos classificados. Nesta implementação a classificação se dá em duas etapas, conforme figura 2.23:

- Extração dos dados – nesta etapa há o tratamento dos dados e seleção da característica escolhida para definir o comportamento; e
- Classificação do comportamento – etapa em que o algoritmo de aprendizagem decide o comportamento. Para isso foram utilizados os algoritmos K-*Nearest Neighbor* (KNN), J48 *Decision Tree* e *Support Vector Machine*, com intuito de oferecer múltiplas saídas de classificação, que ao serem combinadas, melhoram a acurácia do modelo.

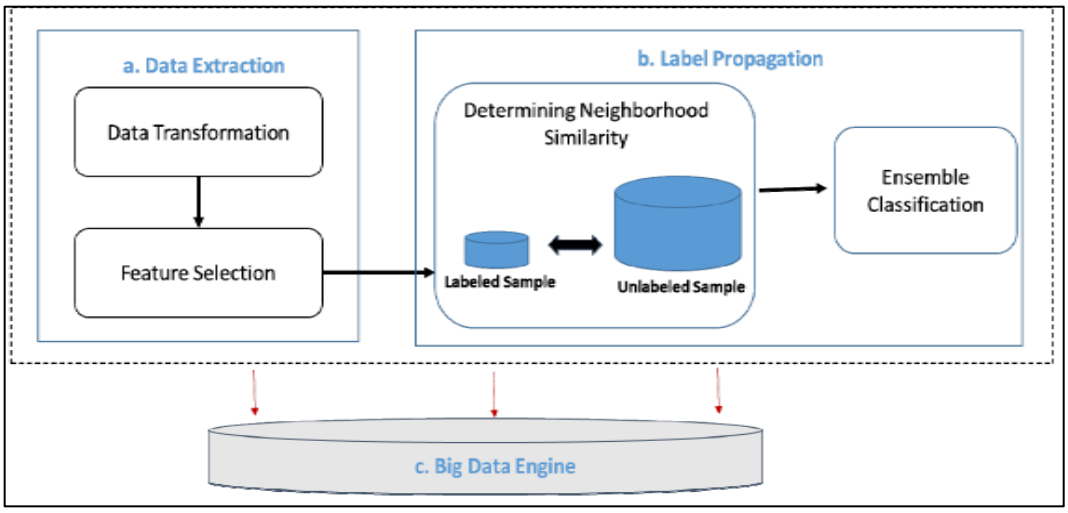



Figura 2.23: Modelo de classificação de fluxo (Pallaprolu et al., 2016)

Na etapa de coleta de tráfego, eles utilizaram duas capturas com variantes do Gh0st RAT. Os dados da captura estão na tabela 2.8. Para extrair as informações dos pacotes da captura, foi utilizado o pySpark em conjunto com o Wireshark CLI, pois, conforme o autor, ferramentas de análise de pacote ainda não são capazes de tratar grande volumes de dados e faltam integração com ferramentas de *big data*. Assim utilizou o pySpark a fim de inserir os dados no Apache Spark.

Tabela 2.8: Dados da captura com Gh0st RAT (Pallaprolu et al., 2016)

| | RAT Dataset 1 | RAT Dataset 2 |
|---|---|---|
| Total Traffic | 779 | 1006 |
| Count for RAT Attack Traffic | 9 | 79 |
| Count for RAT Normal Traffic | 770 | 927 |

Diferentemente do que vinha sendo feito por outros autores, Pallaprolu et al. (2016) preferiu utilizar outros campos dos pacotes, tabela 2.9, e assim definir as características necessárias para classificar o fluxo.

Tabela 2.9: Outras características de detecção utilizadas (Pallaprolu et al., 2016)

| Rank of attribute | RAT Dataset 1 | RAT Dataset 2 |
|---|---|---|
| 1 | Window scale factor | iRTT |
| 2 | Window scale value | Time since reference of 1st frame |
| 3 | Time | Window scale value |
| 4 | Capture length | Window scale factor |
| 5 | Calculated(WSV) | protocol |
| 6 | Acknowledgement number | checksum |
| 7 | Source address | offset |
| 8 | Destination address | Source address |
| 9 | Next sequence number | Destination address |
| 10 | Label(Cybereye, Normal) | Label(Lurk0, Normal) |

A arquitetura desse modelo de detecção elaborado pode ser vista na figura 2.24. Neste caso, a ferramenta de *big data* implementada foi o Apache Spark, devido ao fato de oferecer processamento em memória, o que aumenta o desempenho para grandes volumes de dados, além de possuir característica de ser distribuído.

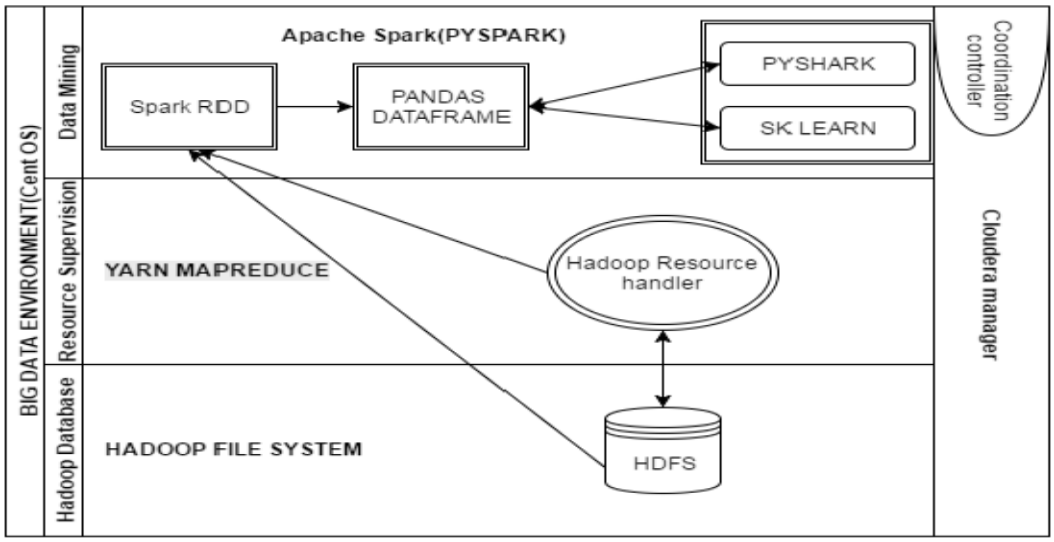



Figura 2.24: Arquitetura de detecção com Spark (Pallaprolu et al., 2016)

Nesta arquitetura, tem-se a parte chamada *data mining* que oferece o processamento do *dataset*, com os módulos pyShark para extrair as características da captura, Sk Learn para executar os algoritmos de *machine learning* e o Pandas para coordenar os dados para o Spark RDD.

Os dados são distribuídos com o Spark RDD para processamento em um cluster gerenciado pelo Yarn e, por fim, os dados são armazenados e consultados no *Hadoop Distributed File System* (HDFS). Foram utilizados os seguintes equipamentos: quatro nós de cluster com 2.10 GHZ Intel Xeon E5-2450, com processador *core* I7, 12 GB de memória, 1 TB 7200 RPM, e sistema operacional CentOS 6.6.

Interessados em verificar o desempenho do Spark se comparado com o Hadoop em relação ao tempo de execução, eles replicaram o *dataset* 1 por 10, 100 e 1000 vezes, a fim de simular grande volume de dados, uma vez que o conjunto de dados inicial era pequeno. Assim, na figura 2.25, percebe-se o desempenho das duas ferramentas.

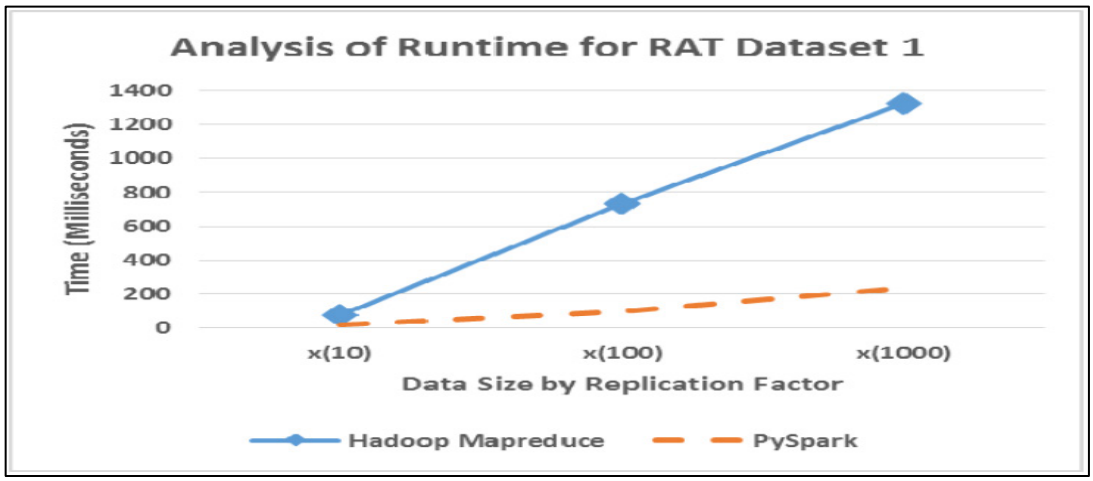



Figura 2.25: Comparação de desempenho entre Spark e Hadoop (Pallaprolu et al., 2016)

O resultado do sistema de detecção para o *dataset* 1 encontra-se na figura 2.26. Com o intuito de validar o objetivo inicial de classificar grandes fluxos com pequenos dados classificados, eles subdividiram o *dataset* 1 em três conjuntos de testes: pequeno, médio e

grande com tamanhos, respectivamente, de 200, 569 e 769 pacotes. Todos com números de pacotes de treinamento para o algoritmo de *machine learning* igual a 10, sendo que 40% dos pacotes eram do RAT e 60% de aplicações normais.

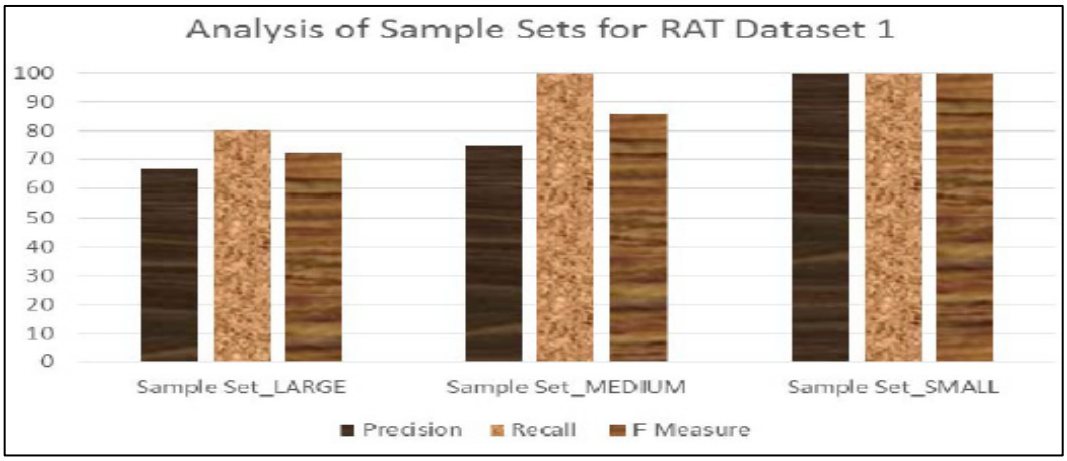



Figura 2.26: Resultado do *dataset* 1 (Pallaprolu et al., 2016)

Nota-se que o modelo proposto por Pallaprolu et al. (2016) reduz o desempenho conforme se aumenta o conjunto de dados para avaliação. Por outro lado, apresentou 100% de detecção no caso do *dataset* 1 pequeno.

Agora o resultado do dataset 2, figura 2.27, não obteve a mesma performance, isso pode ser explicado devido à diferença de características adotadas em cada *dataset*. E também pelo fato dos subconjuntos do *dataset* 2 terem tamanho de testes diferentes do *dataset* 1, possuindo 996, 769 e 200 para o conjunto grande, médio e pequeno, respectivamente. O tamanho e proporção de aplicações no conjunto de treinamento foi o mesmo considerado no *dataset* 1.

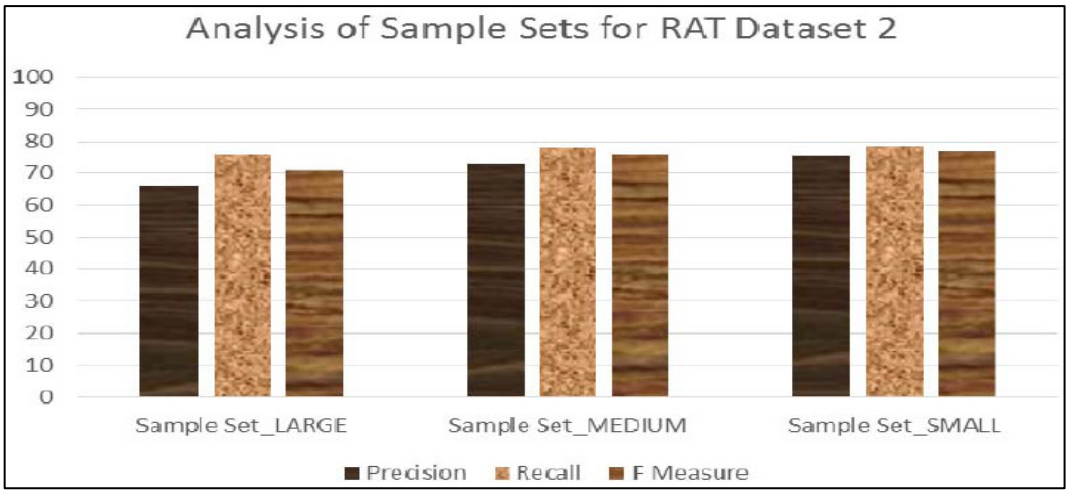



Figura 2.27: Resultado do *dataset* 2 (Pallaprolu et al., 2016)

Esse trabalho apresentou uma forma de se detectar RAT com ferramentas de *big data*, indicando o Spark como uma boa opção para elaborar modelos de detecção com acurácia elevada e excelente desempenho. Porém não utilizou diferentes tipos de RAT para validar o resultado mostrado, bem como não fez aplicação em ambiente real, limitando-se a capturas disponibilizadas por terceiros.

# Capítulo 3

# Projeto e Implementação

Este capítulo abordará o sistema de detecção proposto, a partir do problema explicitado no capítulo 1, tendo como base teórica todas as pesquisas, conceitos e ferramentas apresentadas durante o capítulo 2.

## 3.1 Definição das características dos RAT

Inicialmente, foi idealizado utilizar todas as características resultantes da estatística de conexão do Tshark, figura 3.6, assim como adicionar novas elaborando combinações entre essas características, e também extrair porcentagem de PUSH e média do valor do TTL, a partir do CSV campos de pacotes, mencionado na figura 3.10.

Contudo, o tempo de execução requerido para extrair as características do CSV campos de pacotes foi superior ao tempo de detecção proposto de cinco minutos.

Portanto, as características adotadas para este projeto serão as constantes na tabela 3.1. A característica TS/Duracao retrata a taxa envio de segmentos por segundo durante a conexão, enquanto a característica TB/Duracao a taxa de bytes enviados por segundo, e as características BES/SES, BEC/SEC, BES/BEC e SES/SEC foram combinações feitas para melhor caracterizar o comportamento das conexões.

Tabela 3.1: Características dos RAT para o projeto

| Acrônimo | Característica |
|---|---|
| SES | Total de segmentos enviados pelo servidor |
| BES | Total de bytes enviados pelo servidor |
| SEC | Total de segmentos enviados pelo cliente |
| BEC | Total de bytes enviados pelo cliente |
| TS | Total de segmentos enviados na conexão |
| TB | Total de bytes enviados na conexão |
| Duracao | Duração da conexão |
| TS/Duracao | Razão entre TS e duração da conexão |
| TB/Duracao | Razão entre TB e duração da conexão |
| BES/SES | Razão entre BES e SES |
| BEC/SEC | Razão entre BEC e SEC |
| BES/BEC | Razão entre BES e BEC |
| SES/SEC | Razão entre SES e SEC |

Porém outras combinações podem ser criadas para esse fim, como por exemplo, BES/Duracao ou SES/TS. Portanto cabe outro estudo para verificar a melhor combinação possível para essas características, mas este projeto procurou replicar as combinações estudadas por Jiang e Omote (2015), tabela 2.3. As características do trabalho desses autores possuem equivalência, respectivamente, com as seguintes características: TS, BES, SES, BEC, SEC, SES/SEC e BES/SES.

Além dessas, foram consideradas as combinações criadas por Li et al. (2012), que são duração, total de bytes, total de segmentos e intervalo médio de envio de segmentos da conexão, sendo respectivamente neste projeto: Duracao, TB, TS e TS/Duracao. As demais características - TB/Duracao, BEC/SEC e BES/BEC – foram adicionadas seguindo o raciocínio desses trabalhos relacionados.

## 3.2 Tipos de RAT analisados

O sistema de detecção apresentado pretende identificar qualquer tipo de RAT sem conhecer previamente seu funcionamento, para tal, a abordagem utilizada é de classificar os fluxos de conexões, baseando-se no comportamento de um grupo de treino de RAT.

Assim foram selecionados cinco tipos de RAT para treinar os algoritmos e mais cinco tipos diferentes para testar o desempenho do modelo. A tabela 3.2 exibe os *malwares* escolhidos neste projeto, para efetuar a captura não foi feita alteração dos parâmetros do servidor RAT, portanto usou-se a configuração padrão desses programas.

Tabela 3.2: RAT selecionados para treino e teste

| Aplicação | RAT | Porta Cliente |
|---|---|---|
| Treinamento | Bandook v1.36 | 1167 |
| | Cerberus RAT 1.03.5 | 5150 |
| | Poison Ivy 2.3.2 | 3460 |
| | ProRat v2.1 | 5110 |
| | Turkojan v3.0 | 15963 |
| Teste | Apocalypse v1.4.4 | 1453 |
| | Indetectables Rat v.0.9.5 | 8080 |
| | Optix Pro v1.33 | 3410 |
| | Pandora RAT v2.2 Beta | 6622 |
| | Schwarze Sonne RAT 2.0 | 1515 |

Observou-se que após iniciar o servidor do RAT na máquina da vítima, ele tentou se conectar ao cliente, na porta determinada durante a configuração do servidor, enquanto não conseguia a conexão, era incrementado em um o valor inicial da porta do servidor; e em seguida, outra tentativa era efetuada com essa nova porta de origem, conforme figura 3.1; e no caso de nova conexão, também será incrementado o valor da porta do servidor.

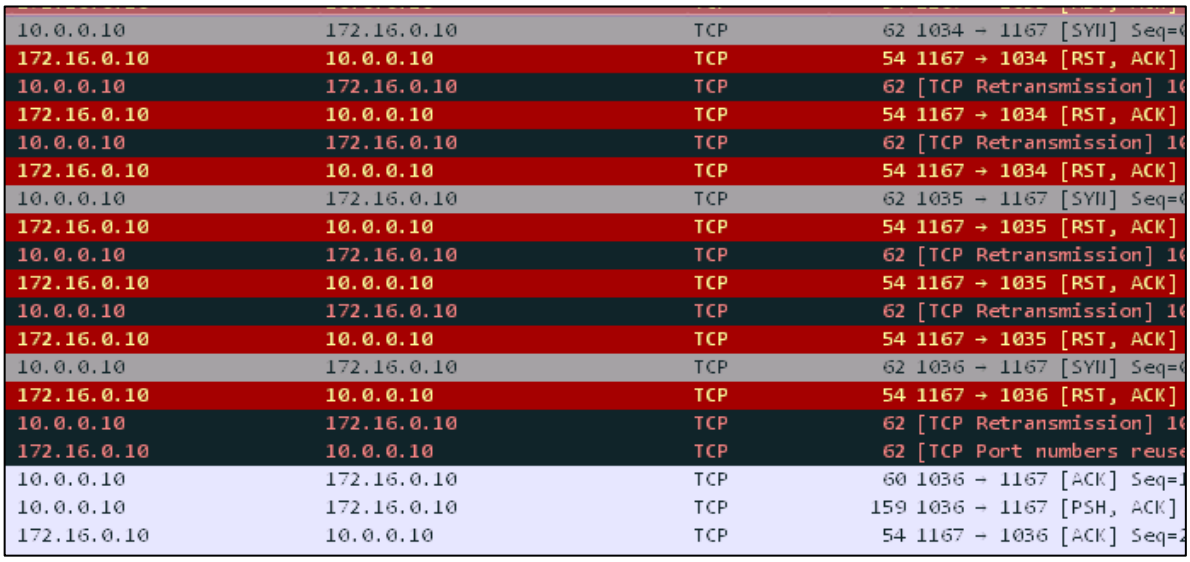



Figura 3.1: Incremento porta servidor RAT

Assim percebe-se um padrão de comportamento dos RAT, pois todos observados possuem a ação de incrementar o valor da porta do servidor, cujo valor inicial utiliza a faixa registrada de portas TCP: 1.024 a 49.151. Durante os testes, de forma geral, notou-se que o último valor é armazenado, e caso o servidor seja reiniciado, a sequência será iniciada com o incremento desse valor.

Ao final ficaram disponíveis para treinamento 15 capturas com 290 conexões dos RAT, e para treino foram 15 capturas com 161 conexões. Essa diferença se deu pela implementação de cada RAT, pois alguns RAT utilizam conexões secundárias para executar a comunicação, isso pode ser algo que poderá impactar no desempenho do sistema de detecção.

## 3.3 Modelo proposto

Após pesquisas na literatura, foi possível identificar como foram projetados os modelos de detecção que utilizam *machine learning*, como por exemplo, o ilustrado na figura 2.16. Conclui-se que basicamente todos os modelos têm em comum os estágios exibidos na figura 3.2.

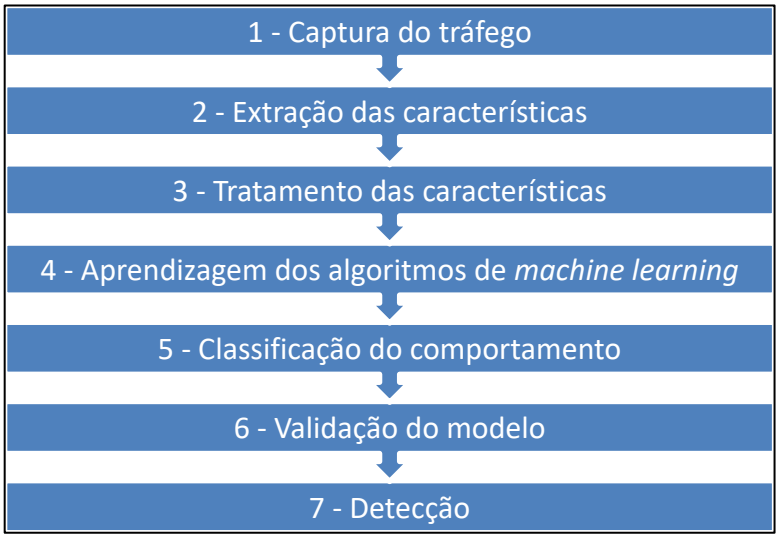



Figura 3.2: Estágios dos sistemas de detecção

Portanto, para criar o modelo deste projeto, serão observados esses estágios, aplicando métodos e ferramentas recentes para análise e detecção dos RAT.

### 3.3.1 Captura do tráfego

A figura 3.3 mostra o estágio de captura do tráfego e sua respectiva ferramenta associada, bem como uma breve descrição das entradas e saídas.

Na fase 1, pretende-se executar uma captura contínua das conexões, e a cada cinco minutos fornecer um arquivo com extensão PCAP com os fluxos capturados. Definiu-se esse tempo, pois conforme Li et al. (2012) em seu estudo, esse foi o tempo limite utilizado para detectar um RAT, assim será mantido esse critério.

Neste estudo será considerado apenas RAT com conexão TCP, da mesma maneira que as pesquisas mais recentes nesse assunto, por isso será removido todos os pacotes que não contenham o protocolo TCP.

Para realizar esta fase será utilizada a ferramenta Tshark para efetuar a captura do tráfego, o comando utilizado foi:

**tshark –f tcp -i (interface do computador) -b duration:300 –b files:2 -w captura.pcapng**

Observa-se nesse comando que a cada cinco minutos é gerado um arquivo de captura, após esse tempo é criado outro arquivo, sem interromper a captura. Foi definido o limite de dois arquivos, porque enquanto um arquivo é analisado pelo sistema de detecção, o outro armazena o tráfego.

Esse comando sobreescreve a captura mais antiga, logo o tempo de execução do sistema deve ser inferior ao tempo de cinco minutos para que se possa analisar todos os arquivos.

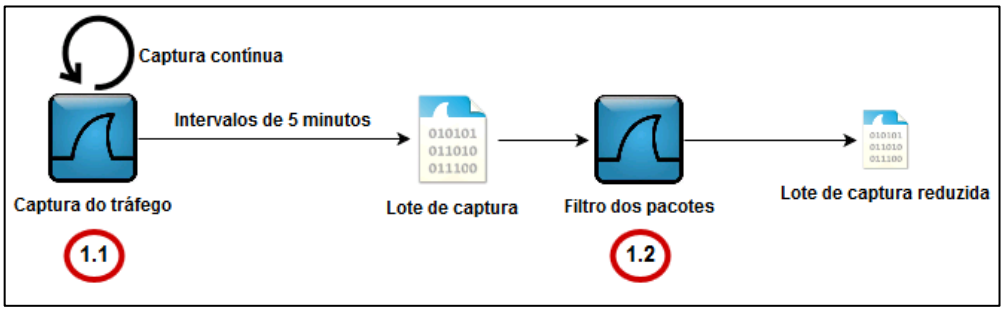



Figura 3.3: Captura do tráfego

Agora na fase 2, foi planejado reduzir o custo computacional do processamento das capturas, a fim de propiciar menor tempo de execução. Como este modelo pretende ser aplicado a redes com grande tráfego de dados, mesmo com um intervalo de cinco minutos, a captura pode ter tamanho de gigabytes.

Li et al. (2012) em seu projeto utilizaram um arquivo de captura de terceiros com tamanho de segmentos igual a 96 bytes, entretanto isso não afetou o desempenho de seu sistema de detecção, pois as características que foram observadas não dependiam do que havia além desse total de bytes. De forma semelhante, Wu et al. (2017) também reduziram o segmento para o tamanho máximo de 95 bytes.

Como foi visto no capítulo 2, quando se utiliza o TCP/IP, cada camada da arquitetura contém um conjunto de protocolos, e estes por sua vez, inserem cabeçalhos aos dados a serem transmitidos. Este trabalho extrairá características das conexões contidas nesses cabeçalhos, portanto não irá considerar os campos de opções que adicionam informação nos protocolos, logo o tamanho do segmento a ser analisado terá 58 bytes, que é o mínimo dos cabeçalhos do IEEE 802.3, IP e TCP.

Então, essa fase, denominada de filtro dos pacotes, propõe reduzir o tamanho total do arquivo de captura original, principalmente para retirar a carga útil que não será analisada. Para executar essa atividade será usado o editcap, sua documentação está disponível em Wireshark (2017), o comando executado é:

**editcap -s 58 captura_original.pcapng  captura_reduzida.pcapng**

A utilização desse comando pode reduzir, em média, aproximadamente 87% do tamanho do arquivo, de acordo com a tabela 3.3, a qual demonstra a relação entre a quantidade de pacotes da captura e o tamanho total da captura, comparando seis capturas de tamanhos e quantidades de pacotes distintas com suas respectivas capturas reduzidas. A figura 3.4 ilustra graficamente essa redução.

Tabela 3.3: Comparativo entre captura original e reduzida

| Quantidade de pacotes | Tamanho original | Tamanho reduzido | Percentual de redução |
|---|---|---|---|
| 309.521 | 226 MB | 28 MB | 87,61% |
| 2.093.782 | 1,251 GB | 192 MB | 84,65% |
| 8.799.763 | 6,507 GB | 809 MB | 87,57% |
| 16.219.295 | 12 GB | 1.491GB | 87,58% |
| 42.503.338 | 34 GB | 3,909 GB | 88,50% |
| 76.627.662 | 59 GB | 7,048 GB | 88,05% |
| 100.000.000 | 72 GB | 9,199 GB | 87,22% |
| **Média do percentual de redução** | | | **87,31%** |

Nesse teste realizado, percebe-se que com capturas a partir dois milhões de pacotes, o processamento do arquivo original demandará elevado custo computacional, e consequentemente alto tempo de execução, porque produz um arquivo com tamanho acima de 1 GB.

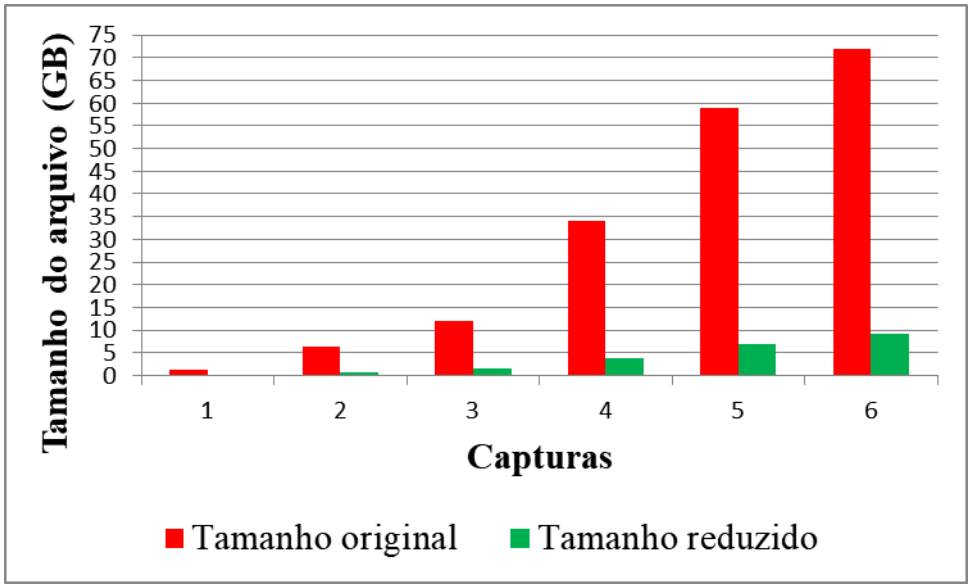



Figura 3.4: Redução do tamanho de captura

Assim, faz-se necessário reduzir o tamanho do arquivo de captura, a fim de tornar viável a detecção do RAT com o critério de intervalos de verificação e baixo custo computacional. Além disso, irá gerar pacotes com tamanho fixos, sem perder as principais informações, o que facilitará a análise de tempo de execução, pois os pacotes das capturas a cada intervalo de tempo tendem a ter tamanhos diferentes.

### 3.3.2 Extração das características

Continuando com a descrição do modelo, o próximo estágio é a extração das características da captura que foi reduzida, figura 3.5. Como a captura gera arquivos com extensão PCAP e as aplicações de *big data*, pelo menos até onde foi pesquisado,

não conseguem receber como entrada esse tipo de arquivo, portanto é necessário ler a captura para extrair as informações, definidas como relevantes, para o sistema de detecção.

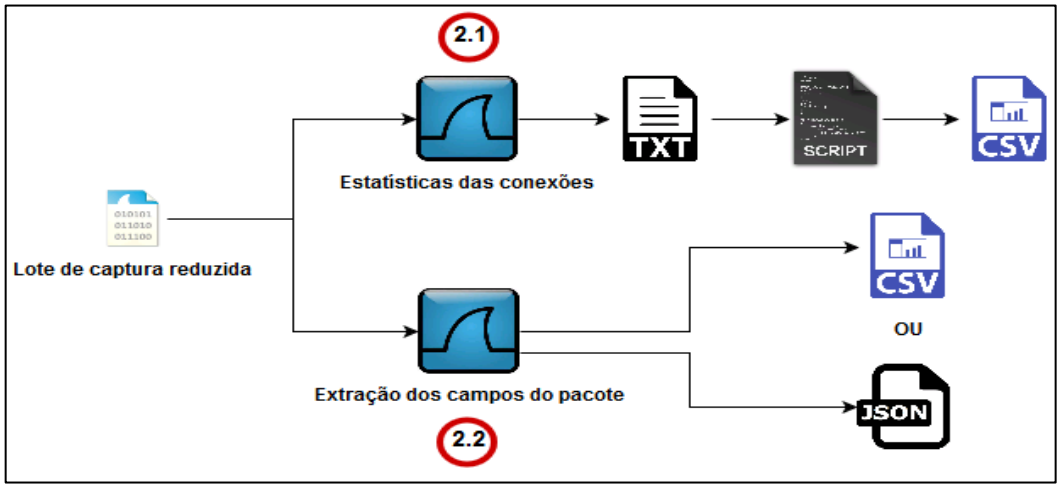



Figura 3.5: Procedimento de extração das características

Ao analisar a documentação do Tshark, verificou-se que há um tipo de estatística gerada muito útil para retirar informações da captura. Dado que o objeto de estudo é o comportamento das conexões, essa estatística fornece diversas características das conexões estabelecidas. Portanto a primeira fase desse estágio é obter essa estatística, para tal, o seguinte comando é utilizado:

**tshark -q -z conv,tcp -r captura_reduzida.pcapng > arquivo_estatisticas**

Porém a saída desse comando é um relatório em formato de texto, figura 3.6, então há a necessidade de se tratar essa saída para ser utilizada por uma aplicação. No apêndice B tem o *script* elaborado para formatar essa estatística.

```
================================================================================
TCP Conversations
Filter:<No Filter>
                                               |       <-      |  |       ->      |  |     Total     |    Relative    |   Duration   |
                                               | Frames  Bytes |  | Frames  Bytes |  | Frames  Bytes |      Start     |              |
172.16.0.10:3460    <-> 10.0.0.10:1073           245    224130     198    34821      443    258951    44.157756452     358.0987
172.16.0.10:3460    <-> 10.0.0.10:1074            12      1230      15    15915       27     17145    377.551560729     24.7052
================================================================================
```

Figura 3.6: Saída Tshark – z conv,tcp

A figura 3.7 tem a finalidade de verificar a curva de crescimento dos arquivos de saída, após o tratamento e formatação em CSV pelo *script*. Observa-se que à medida que há aumento da quantidade de pacotes capturados, maior a chance de se capturar conexões distintas, e como o comando faz estatísticas das conexões, armazenando em cada linha uma conexão, o arquivo tenderá aumentar proporcionalmente a quantidade de conexões capturadas.

Outro fato importante a se considerar, é o tamanho total no caso de uma captura de 9 GB, que resultou em arquivo de saída de apenas 85 MB, o que demonstra ser uma bom método de extração.

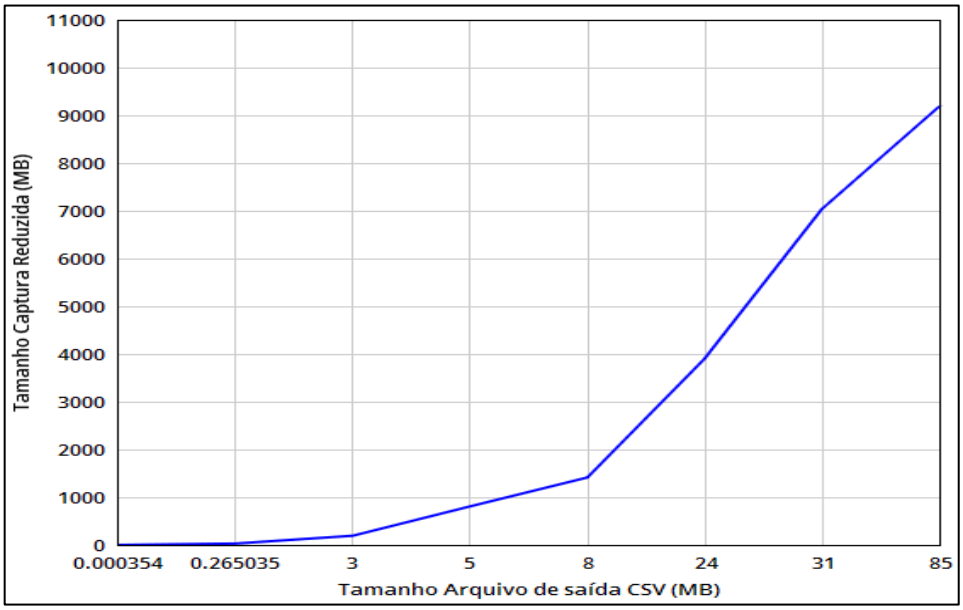



Figura 3.7: Curva de crescimento dos arquivos de saída do comando conv,tcp

A segunda fase deste estágio é complementar, pois dependendo das características escolhidas para modelar o comportamento no sistema de detecção, a saída ilustrada na figura 3.6 será suficiente, como é o caso do sistema implementado por Jiang e Omote (2015), o qual as características, tabela 2.3, podem ser derivadas usando esse relatório.

Entretanto neste projeto foi idealizado utilizar outras características que não são contempladas por essa saída. Considerando esse objetivo, há outro comando do Tshark que possibilita extrair qualquer campo de todos os pacotes, oferecendo como saída diversos formatos, tais como JSON, texto e outros.

O comando utilizado para esse fim pode ser visto no apêndice B, o qual demonstra o código para exportar alguns campos dos pacotes para os formatos CSV, JSON e JSON Elasticsearch. A figura 3.8 exibe a saída desse comando para o formato JSON, os campos extraídos foram: tempo relativo da captura do pacote, *index* de identificação do fluxo TCP (TCP stream) e o estado da *flag push* do TCP.

```
,
{
  "_index": "packets-2017-11-09",
  "_type": "pcap_file",
  "_score": null,
  "_source": {
    "layers": {
      "frame.time_relative": ["44.157387770"],
      "tcp.stream": ["0"],
      "tcp.flags.push": ["0"]
    }
  }
}
```


Figura 3.8: Extração de campos dos pacotes para JSON

Como as ferramentas de *big data* aceitam nativamente o JSON, seria o candidato ideal para formato de saída, porém como pode ser visto na figura 3.9, esse formato tem um custo.

O tamanho do arquivo JSON gerado tende a crescer muito mais rápido do que o formato CSV, portanto também neste caso, deve-se atentar para o tempo de execução necessário para concluir essa tarefa, pois quando o arquivo é maior, geralmente, requer maior tempo de execução.

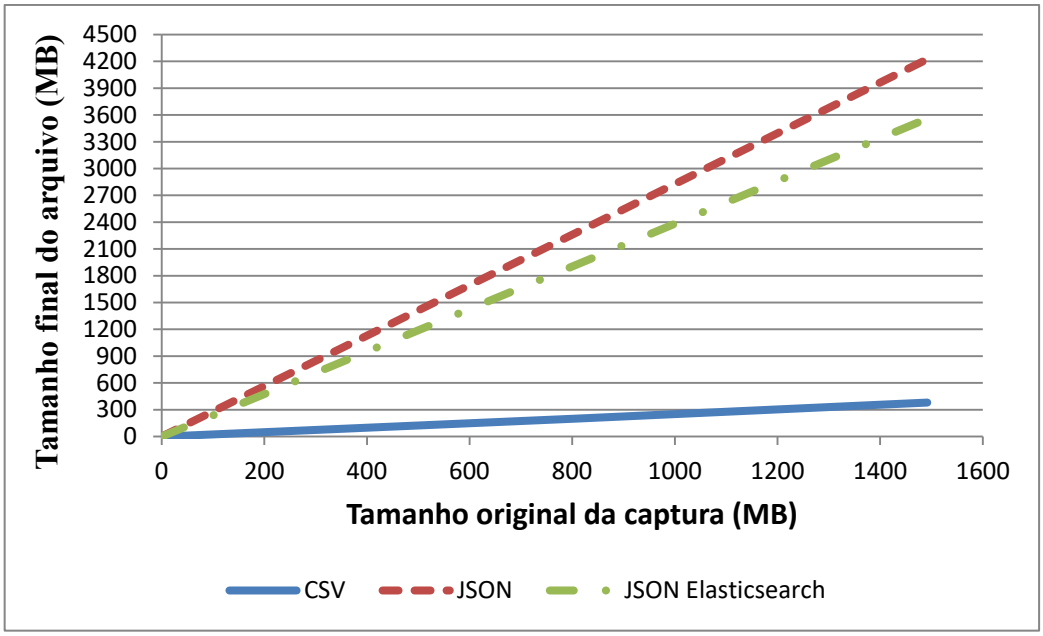



Figura 3.9: Comparativo entre tamanho de saída dos formatos CSV e JSON

Percebe-se na figura 3.9, que a escolha do formato JSON pode provocar um aumento de aproximadamente onze vezes do arquivo de saída, quando comparado ao CSV. Isso pode se tornar um grande problema, principalmente, em capturas acima de 400 MB, pois os arquivos de saída passaram de 1 GB.

Observa-se que a saída em JSON cria arquivos, aproximadamente, 2.7 vezes maiores do que a captura original, assim em todos os casos há aumento do tamanho do arquivo a ser utilizado, entretanto isso não acontece com o CSV.

Esse aumento se deve ao fato do JSON acrescentar campos para cada atributo, com a finalidade de estruturar o arquivo, enquanto o CSV utiliza apenas um cabeçalho para identificar cada campo e os separa com uma vírgula, logo o tamanho do arquivo ficará menor quando comparado ao JSON.

Como o Apache Spark pode receber como entrada tanto o JSON quanto o CSV, optou-se por utilizar o CSV mediante a esses benefícios.

### 3.3.3 Tratamento das características

O próximo estágio é o tratamento das características extraídas da captura filtrada. Neste caso será utilizado o módulo SQL do Apache Spark, pois utiliza linguagem apropriada para manipular arquivos tabulados. A figura 3.10 exibe o processo de tratamento planejado.

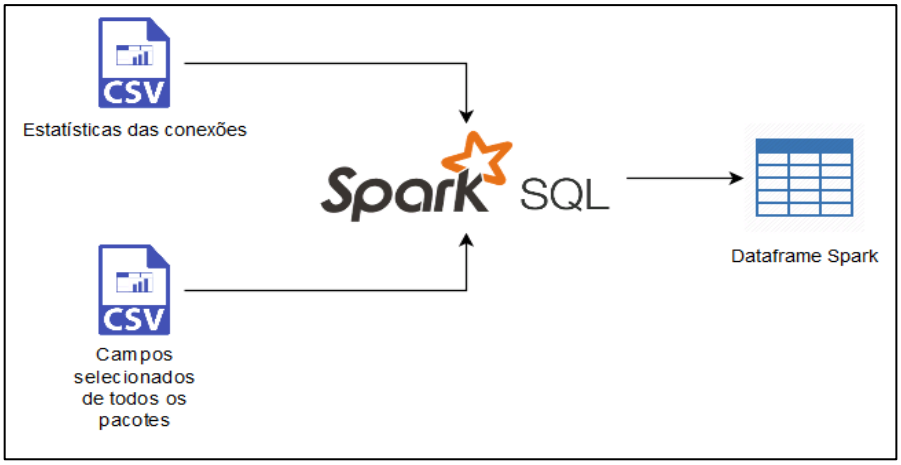



Figura 3.10: Tratamento dos arquivos CSV

No apêndice C pode ser visto o código criado para tratar os dois arquivos de entrada no formato CSV, e no apêndice D tem-se a classe com a configuração dos componentes necessários para iniciar o Spark.

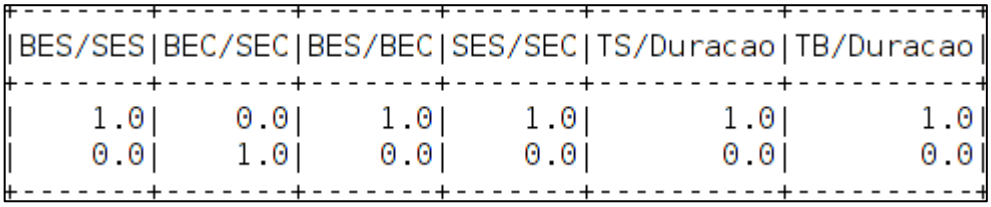
Após executar o código de tratamento do CSV, tem-se um dataframe Spark conforme a figura 3.11, contendo as características extraídas do arquivo Estatísticas das conexões, que foram normalizadas de acordo com a equação 1. Como há somente duas conexões no arquivo de teste usado, os valores normalizados ficaram zero ou um, mas ao aplicar esse método, os valores ficam contínuos entre zero e um.

```
+--------------+---------+-------+-----------+-------+---+---+---+---+---+---+-------+
|Tempo_Relativo| Servidor|S_Porta|    Cliente|C_Porta|SES|BES|SEC|BEC| TS| TB|Duracao|
+--------------+---------+-------+-----------+-------+---+---+---+---+---+---+-------+
|  44.157756452|10.0.0.10|   1073|172.16.0.10|   3460|1.0|1.0|1.0|1.0|1.0|1.0|    1.0|
| 377.551560729|10.0.0.10|   1074|172.16.0.10|   3460|0.0|0.0|0.0|0.0|0.0|0.0|    0.0|
+--------------+---------+-------+-----------+-------+---+---+---+---+---+---+-------+
```

Figura 3.11: Dataframe Spark com estatísticas do Tshark

Continuando com as colunas do dataframe, a figura 3.12 contém a combinação criada entre as colunas, com intuito de adicionar novas características. A descrição

dessas características testadas está comentada no código do apêndice C e na seção 3.1, que versa sobre as características utilizadas neste projeto.

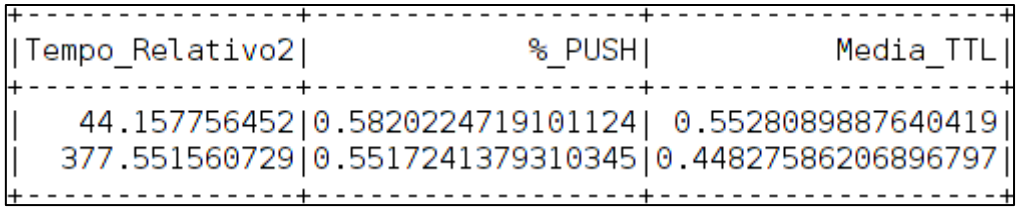


| BES/SES | BEC/SEC | BES/BEC | SES/SEC | TS/Duracao | TB/Duracao |
|---|---|---|---|---|---|
| 1.0 | 0.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 |

Figura 3.12: Dataframe Spark com colunas combinadas

Em seguida, foi idealizado testar outras características, além das disponíveis na estatística do Tshark. Assim foram extraídos os campos TCP.FLAG.PUSH e IP.TTL, o primeiro foi relatado por Yamada et al. (2015) como um boa opção para identificar o RAT; enquanto o segundo foi planejado como forma de mensurar se o cliente está na rede interna ou não, devido a média da quantidade de saltos. Sabe-se que o seu valor pode ser alterado e cada sistema operacional possui um número máximo diferente, entretanto buscou-se adicionar outras opções de características.

A figura 3.13 ilustra o dataframe obtido após execução do código de tratamento do CSV com os campos dos pacotes, apêndice C. A coluna "Tempo_Relativo2" foi utilizada como chave estrangeira para manter a correta correlação entre os dois dataframes criados, possibilitando efetuar um *join* para unificar todas as características. Nos comentários do código há uma explicação mais detalhada de como foram agrupados os pacotes e atribuídos as suas respectivas conexões.



| Tempo_Relativo2 | %_PUSH | Media_TTL |
|---|---|---|
| 44.157756452 | 0.5820224719101124 | 0.5528089887640419 |
| 377.551560729 | 0.5517241379310345 | 0.44827586206896797 |

Figura 3.13: Dataframe Spark com campos extraídos dos pacotes

Embora a parte do código que extrai os campos dos pacotes tenha funcionado, apresentou-se tempo de execução extremamente alto, pois com um arquivo com 3255 conexões e 309.521 pacotes levou aproximadamente 50 minutos para terminar a execução. Mesmo configurando o Spark para utilizar oito núcleos de processadores com 2.6 GHZ e memória RAM de 16GB.

Essa parte do código escrito não oferece melhor desempenho para essa aplicação, pois acaba criando um dataframe para cada conexão, com a finalidade de agrupar os pacotes. Seria melhor alterar o código fonte da função "z,conv" do Tshark para acrescentar novos campos para as estatísticas das conexões.

Por isso essa parte do código não será usada no modelo, porque um dos critérios adotados no projeto é atender o tempo limite de execução de cinco minutos.

### 3.3.4 Aprendizagem dos algoritmos de machine learning

Prosseguindo na descrição dos estágios, a aprendizagem dos algoritmos de *machine learning* foi dividida em duas etapas, a primeira utiliza o algoritmo K-*means* para separar as conexões em grupos similares (agrupamento), enquanto a segunda utiliza o *Random Forest* para classificar o fluxo.

A primeira etapa pode ser vista na figura 3.14, durante os testes no modelo notou-se elevada quantidade de erros de agrupamento, quando eram aplicadas todas as características extraídas em apenas uma execução do K-*means*, denominado neste projeto como *clusters*.

Portanto nesta etapa, as características selecionadas foram divididas em grupos de características, que ofereciam menor taxa de erro de agrupamento, para isso foi feita a combinação de todas as características.

Outro padrão percebido foi que acima de três características por grupo, neste projeto, o algoritmo começava apresentar taxa de erro elevada, mesmo utilizando um *dataset* com 1,083955 milhão de conexões.

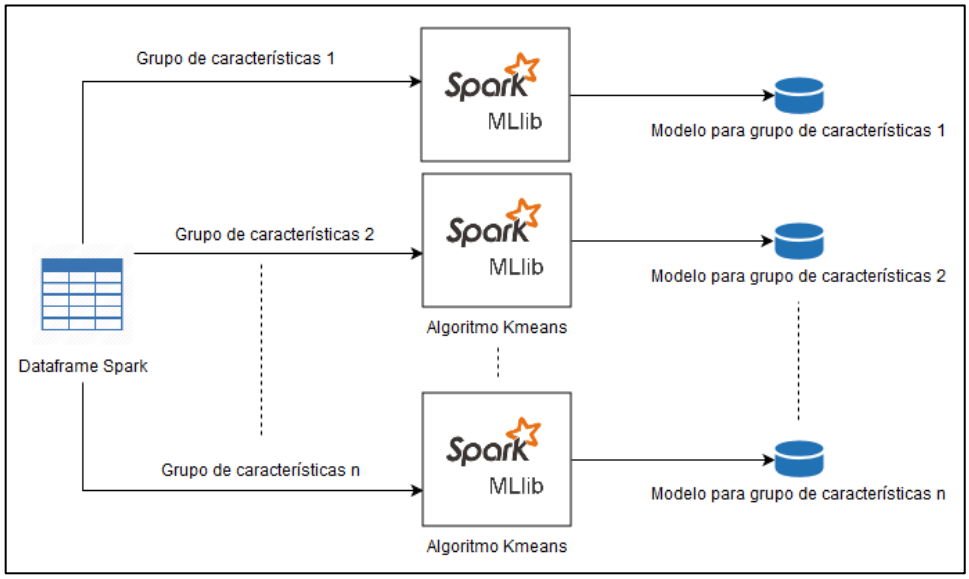



Figura 3.14: Aprendizagem do algoritmo K-*means*

Desse modo, os grupos de características são as entradas para o K-*means* e a saída é o modelo dos grupos identificados na captura, este será salvo em arquivo para agrupar futuramente as novas conexões, conforme suas características.

Uma vez que esse algoritmo precisa como parâmetro de configuração o valor do K, que é o número de *clusters* gerados pelo K-*means*, foi usada a quantidade de portas distintas dos servidores das conexões para definir esse valor. Isso se deve pelo fato de que geralmente um serviço é identificado pela porta do servidor utilizada na comunicação.

No apêndice E, tem-se o código implementado para esta etapa. A figura 3.15 exibe as colunas do *dataframe* gerado com os agrupamentos feitos na saída do K-means. A métrica utilizada para verificar a taxa de erro do modelo K-means é o

*Within Set Sum of Squared Error* (WSSSE), a tabela 3.4 mostra um exemplo de resultado dessa métrica.

Tabela 3.4: Exemplo de taxa de erro do K-*means*

| ***Cluster*** | **WSSSE** |
|---|---|
| 0 | 0.009 |
| 1 | 0.0015 |
| 2 | 0.008 |
| 3 | 0.0026 |
| 4 | 0.018 |

De acordo com o Huguenard LAB (2017), o *Sum of Squared Error* (SSE) é a soma das diferenças entre o dado observado e a média do grupo do cluster, sendo aplicado como medida para mensurar a variação dos valores dentro do cluster. Quanto mais próximo de zero o valor resultante for, melhor será o desempenho do algoritmo K-*means*.

| cluster0 | cluster1 | cluster2 | cluster3 | cluster4 |
|---|---|---|---|---|
| 331 | 54 | 10 | 1510 | 111 |
| 422 | 1 | 123 | 880 | 43 |
| 254 | 1073 | 152 | 280 | 143 |
| 221 | 3 | 87 | 1249 | 87 |
| 1286 | 905 | 684 | 220 | 829 |

Figura 3.15: Dataframe após aplicação do Kmeans

Portanto, além das colunas do *dataframe* exibidas nas figuras 3.11 e 3.12, serão adicionadas as colunas da figura 3.15, com a finalidade de identificar quais os grupos de características a conexão pertence.

Na segunda etapa do estágio de aprendizagem dos algoritmos de *machine learning*, deve-se treinar o *Random Forest* com dados de treinamento gerados após identificação dos grupos, conforme figura 3.16. Assim, além de criar o modelo dos grupos, será criado um modelo geral que incluí os resultados do K-*means*.

Neste caso os valores não precisam ser normalizados, pois o algoritmo *Random Forest* não necessita desse procedimento para classificar os dados. Entretanto foi testado o impacto da normalização, com ou sem normalização das colunas “cluster” o resultado foi o mesmo.

Como esse algoritmo é de aprendizagem supervisionada, há a necessidade de se atribuir rótulos para cada conexão além das características, de acordo com a figura 2.20. Para isso, foi adicionada uma coluna denominada “label” com valores zero ou um, sendo zero para tráfego normal e um para o RAT.

O código desta etapa está no apêndice F, nele é possível visualizar o carregamento do modelo treinado do K-*means*, bem como sua aplicação nos dataframes.

Em seguida, treina-se o *Random Forest* com as colunas de características e os grupos definidos pelo K-*means*. Tem-se como boa prática, separar uma parte do dataframe para treinar e outra para testar o modelo, este teste é parcial, serve apenas para verificar preliminarmente o impacto dos parâmetros.

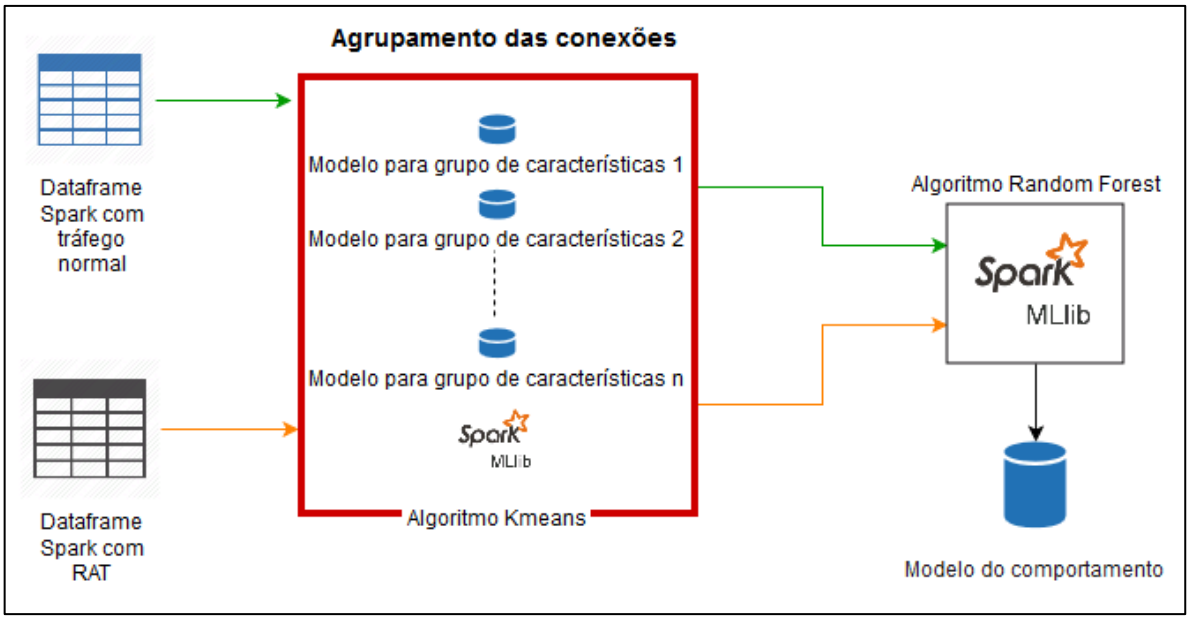



Figura 3.16: Aprendizagem do algoritmo *Random Forest*

O resultado desta etapa é o modelo de comportamento da rede adicionado ao comportamento dos RAT, a partir disso o algoritmo *Random Forest* poderá classificar o fluxo adequadamente.

### 3.3.5 Classificação do comportamento

A seguir, no estágio denominado classificação do comportamento, o *Random Forest* utiliza o modelo aprendido para identificar e classificar o fluxo de acordo com suas características.

O processo é similar ao da figura 3.16, a diferença está no carregamento do modelo salvo no estágio anterior. O código comentado para implementar esse estágio está no apêndice G.

Resumidamente, após carregar o modelo que foi aprendido, o algoritmo classifica as conexões ao atribuir o valor zero ou um na coluna “label”, conforme a conexão de entrada, ou seja, a coluna deverá ter valor um no caso de conexão com RAT, e zero quando for conexão normal. A figura 3.17 exibe o processo desse estágio.

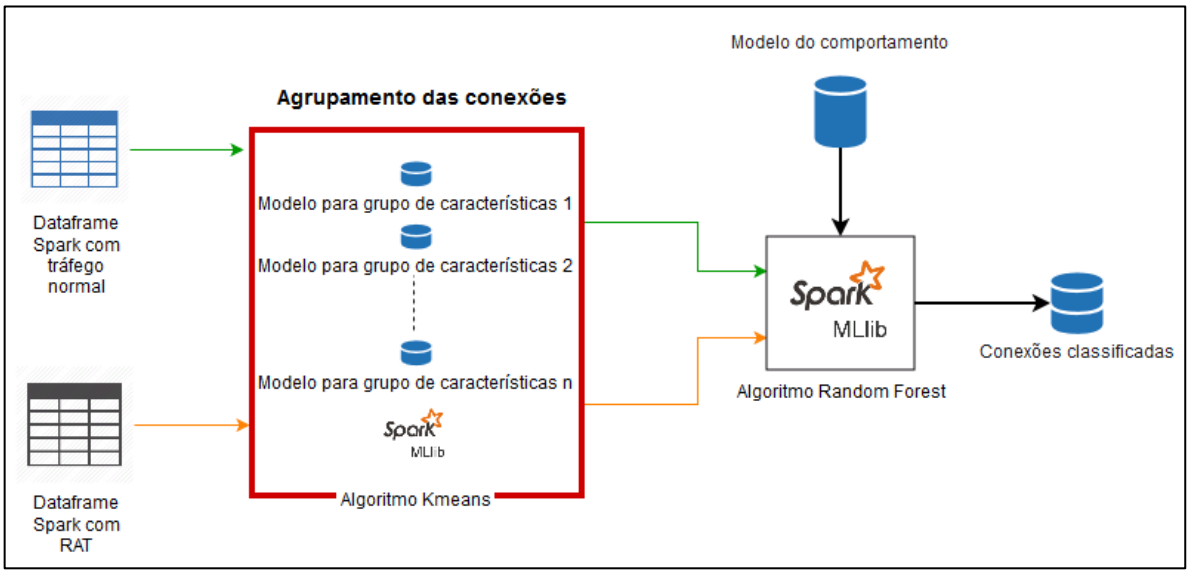



Figura 3.17: Classificação das conexões

O resultado é verificado no estágio de validação do modelo com a utilização da *confusion matrix* e as métricas de desempenho *accuracy*, *precision*, *recall* e F *measure*, as quais estão implementadas no final do código do apêndice G.

### 3.3.6 Validação do modelo

Basicamente, dependendo dos valores das métricas, há a necessidade de ajustar o modelo a fim de melhorar o sistema de detecção, a figura 3.18 ilustra a saída da validação de um modelo que precisa ajustar os parâmetros dos algoritmos K-*means* e *Random Forest*.

A saber, os parâmetros alterados que influenciaram no desempenho do projeto, foram para o K-*means*: o valor do K e o número de interações, e para o *Random Forest*: a profundidade máxima de cada árvore e o número de árvores utilizadas.

```
Confusion Matrix:
2573.0  682.0
116.0   45.0
Accuracy:
0.7663934426229508

Precision(0.0) = 0.9568612867236891
Precision(1.0) = 0.061898211829436035

Recall(0.0) = 0.7904761904761904
Recall(1.0) = 0.2795031055900621

F1-measure(0.0) = 0.8657469717362045
F1-measure(1.0) = 0.10135135135135133
```

Figura 3.18: Resultado da validação

Nota-se que esse modelo exemplificado apresenta elevado número de falso negativo e falso positivo, possuindo, respectivamente, 116 e 682. Esses valores representam a quantidade de conexões classificadas, logo há a necessidade de se alterar os parâmetros dos algoritmos para melhorar a performance, e em último caso elaborar novos agrupamentos de características do K-*means* ou selecionar apenas algumas colunas para a aprendizagem do *Random Forest*.

A figura 3.19 mostra o processo de ajuste dos parâmetros para reduzir os erros de classificação do modelo. Observou-se que esse estágio é o mais sensível e difícil de executar, pois a alteração de um parâmetro impacta diretamente no resultado.

Logo é necessário efetuar diversas combinações entre os parâmetros para ajustar o sistema de detecção, sendo que muitas vezes há um valor limite de cada parâmetro, e caso seja ultrapassado, o desempenho reduz com o fenômeno de *overfit* do modelo. Assim como o valor não pode ser muito inferior, porque gera *underfit* do modelo.

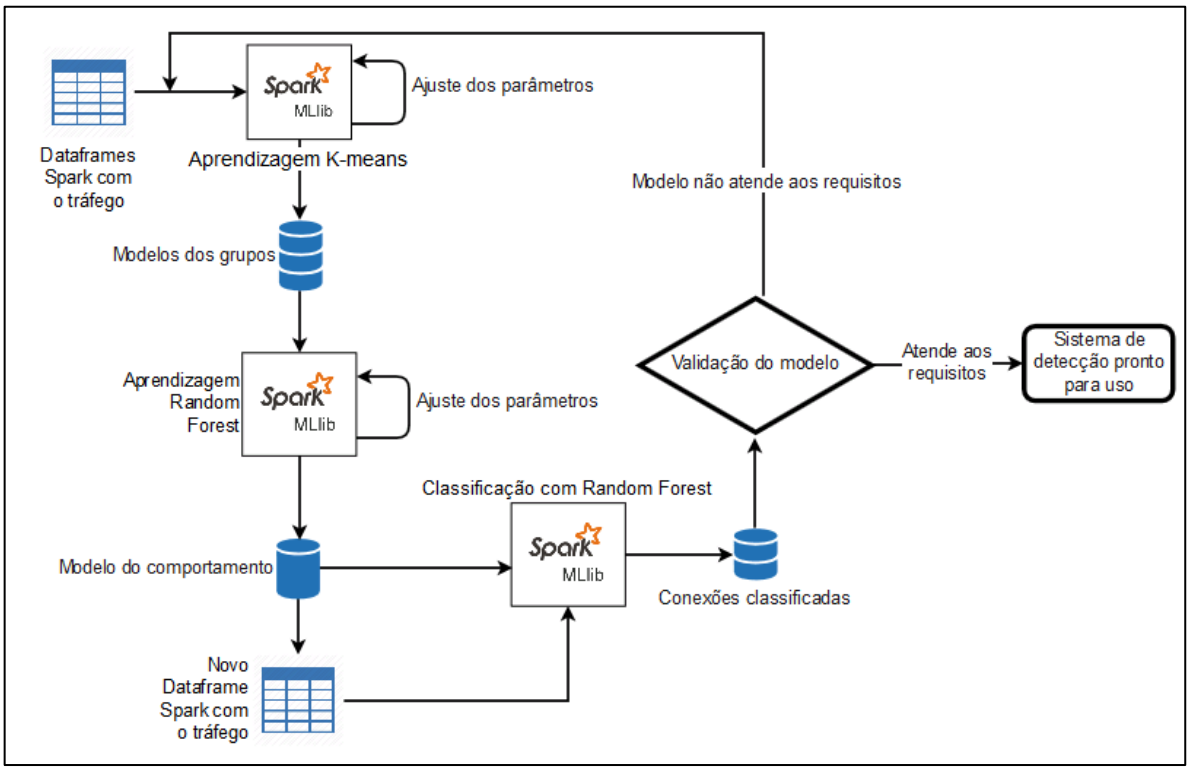



Figura 3.19: Ajuste do modelo

### 3.3.7 Detecção

Por fim, a figura 3.20 exibe uma visão geral do sistema de detecção de RAT com alguns estágios não implementados, mas que fazem parte da proposta do projeto. Após a validação do modelo de detecção é necessário exibir os resultados das detecções de forma automática e fácil, para isso pode-se utilizar ferramentas de análise e exibição de logs.

Como o Spark oferece a possibilidade de trabalhar com JSON, tanto para entrada quanto para saída de dados, há a oportunidade de armazenar o dataframe nesse formato em um banco de dados, com intuito de posteriormente alguma ferramenta faça consulta e atualize um *dashboard web* com as classificações das conexões em intervalos de cinco minutos.

Outro ponto importante para o projeto é armazenar para auditoria, somente aquelas capturas que tenha uma ou mais conexões classificadas com RAT, assim haverá possibilidade de redução do tamanho de armazenamento. Entretanto, caso o modelo apresente baixa acurácia, essa verificação não fará diferença. Este estágio é importante para efetuar uma DPI ou outro tipo de análise de pacotes, a fim de extrair mais informações sobre o ataque.

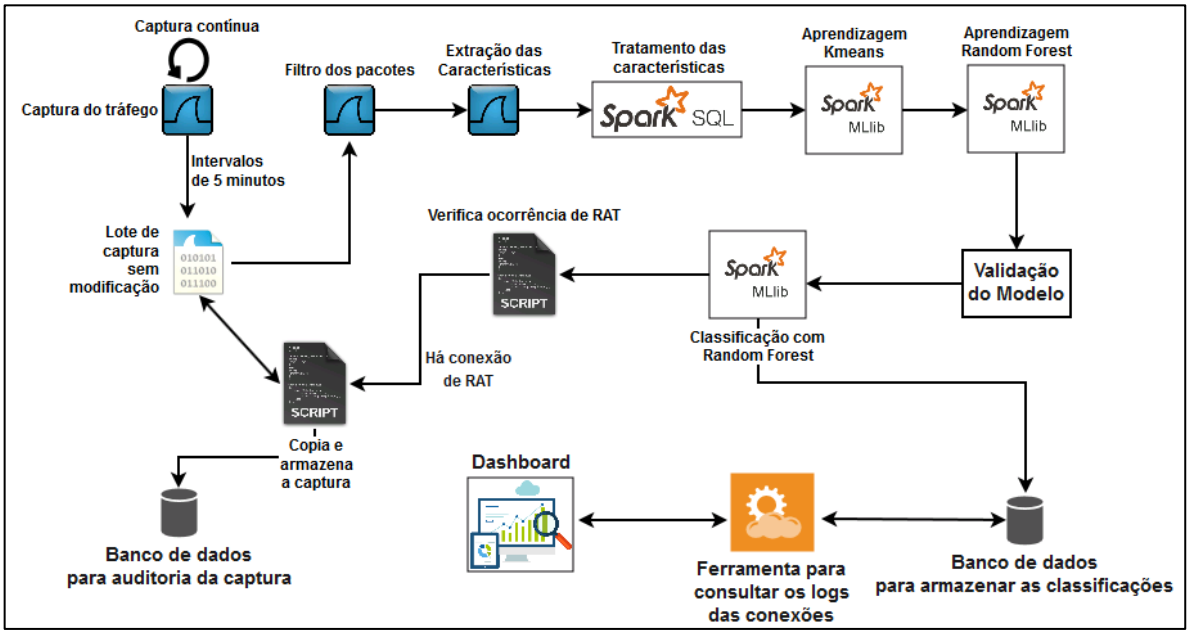



Figura 3.20: Sistema de detecção proposto

Cabe ressaltar que o projeto será implementado até o estágio de validação, e consequentemente, a definição de um modelo para detecção de RAT, logo o restante do modelo poderá ser desenvolvido em projetos futuros.

## 3.4 Desempenho dos elementos do projeto

Esta seção objetiva verificar o tempo de execução de cada estágio do modelo, com intuito de estudar a viabilidade de se implementar o projeto dentro do limite de cinco minutos. Além disso, espera-se determinar o requisito mínimo de *hardware* para executar essa aplicação.

Neste projeto foi utilizado para desenvolvimento e implementação do ambiente Spark, e utilização do Tshark, uma VM com a configuração de acordo com a tabela 3.5, sendo gerenciada pelo hypervisor VMware com virtualização completa.

Tabela 3.5: Configuração máquina virtual para implementação

| Sistema Operacional | Memória RAM | HD | Processadores |
|---|---|---|---|
| Ubuntu Server 64-bit | 24 GB DDR3 1333 Mhz | 120 GB | 8 x Intel(R) Xeon(R) CPU E5-2650 v2 2.6 Ghz |

### 3.4.1 Tshark: extração das características da captura

Como a execução do Tshark se dá por meio do *shell script*, foi necessário utilizar o cgroups, tutorial disponível em Archlinux (2017), para limitar a quantidade de memória RAM usada pelo *script*, e o comando **echo 0 > /sys/devices/system/cpu/cpu<X>/online**, tutorial disponível em NixCraft (2009), para limitar a quantidade de processadores do sistema operacional, a fim de verificar o tempo de execução da aplicação em cada configuração.

Não foi limitada a quantidade de memória virtual para não encerrar o processo por falta de memória, assim caso a memória RAM limitada não seja suficiente, será utilizada a virtual, o que produz aumento do tempo de execução.

A figura 3.21 exibe o gráfico do tempo de execução do Tshark usando memória RAM de 16 MB, variando a quantidade de processadores e utilizando sempre uma captura com tamanho de 28.5 MB, que no ambiente implementado, representa uma captura reduzida de dez minutos de tráfego da rede, o qual contém 309.521 segmentos e 3255 conexões TCP.

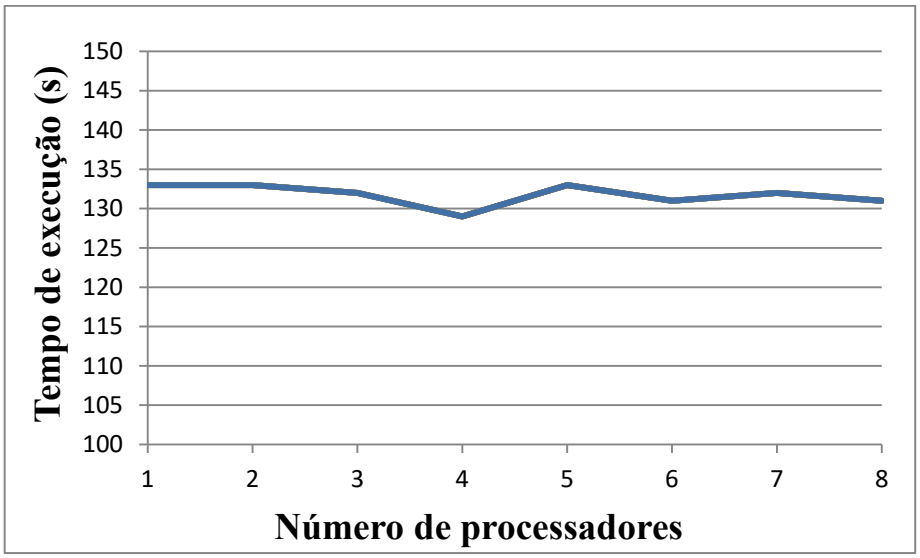



Figura 3.21: Tempo de execução Tshark com 16 MB - alterando número de processadores

O tempo médio de execução com memória de 16 MB foi de 131.75 segundos, percebe-se que aumentar ou diminuir a quantidade de processadores não influencia no tempo de execução do *script*, isso se deve ao fato do Tshark não implementar *multithread* nativamente. Assim a execução desse *software* utiliza apenas um processador.

Portanto, nos testes de tempo de execução do Tshark subsequentes não terá essa variação de processadores, pois é desnecessária. A figura 3.22 mostra o tempo necessário para executar a mesma captura, só que agora será aumentando a memória utilizada pelo *script* em potências de 2.

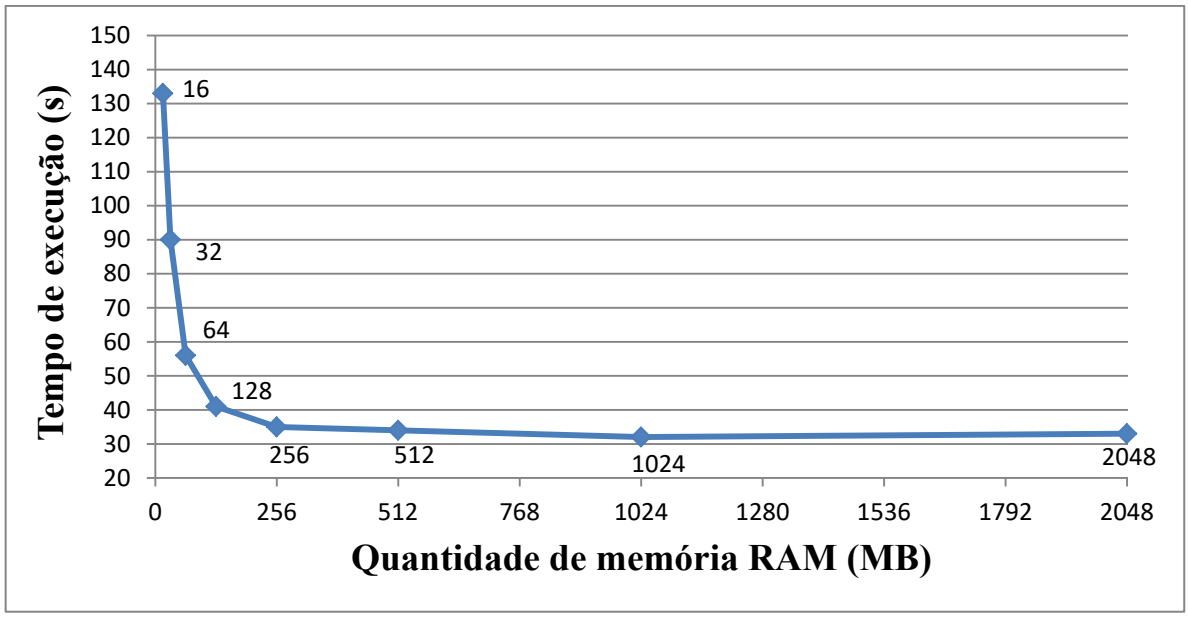



Figura 3.22: Tempo de execução - Tshark

Observa-se que a partir de 1024 MB, o tempo de execução estabiliza em 32 segundos, variando muito pouco com maior quantidade memória, porém com 256 MB o valor foi de 35 segundos, sendo também uma opção viável de se utilizar. Portanto será necessária memória RAM superior a 256 MB para atingir o melhor desempenho nesse estágio.

### 3.4.2 Apache Spark: tratamento das características

Continuando com os testes de desempenho dos estágios do modelo proposto, será avaliada a perfomance do tratamento do CSV estatísticas das conexões gerado pelo Tshark, que neste caso será um arquivo de tamanho igual a 265 KB.

Diferentemente do Tshark, o Spark possui a opção de dedicar quantidade de memória específica para execução de suas tarefas, bem como utilizar *multithread*. O código necessário para limitar memória RAM e número de processadores está no apêndice D, o qual é a classe que define a configuração do Spark.

Ao utilizar 256 MB com memória para o Spark, apareceu a seguinte mensagem de erro: "***ERROR SparkContext: Error initializing SparkContext. java.lang.IllegalArgumentException: System memory 239075328 must be at least 471859200. Please increase heap size using the --driver-memory option or spark.driver.memory in Spark configuration.***"

Essa mensagem significa que o Spark necessita de pelo menos 472 MB para funcionar adequadamente, portanto na figura 3.23 exibe o tempo de execução do tratamento do CSV com memória RAM de 512 MB, variando a quantidade de processadores.

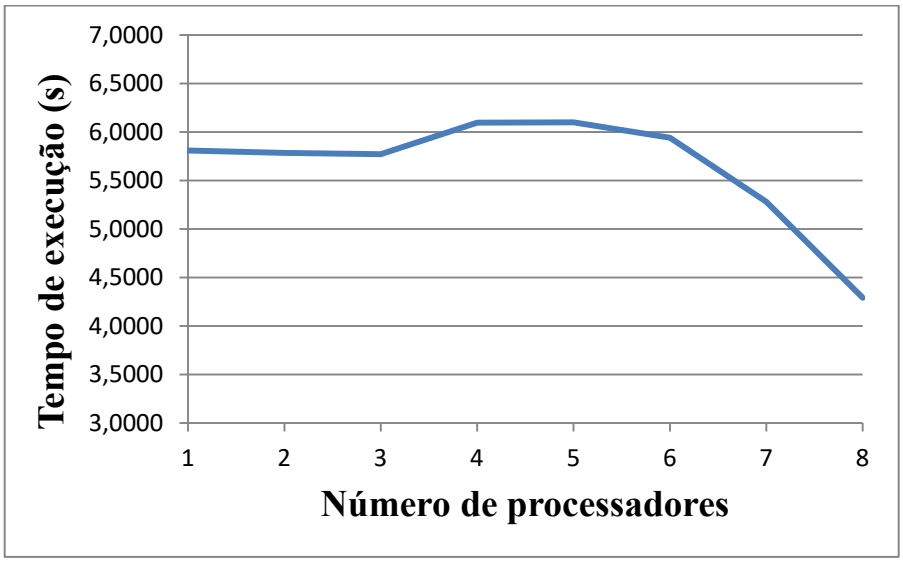



Figura 3.23: Tempo de execução Spark – Tratamento CSV

O tempo de execução da figura 3.23 foi calculado utilizando a opção "load" do spark-shell, ao instanciar a classe pela primeira vez. Porém a partir do segundo "load" da mesma classe, o tempo de execução cai para aproximadamente um segundo, conforme figura 3.24.

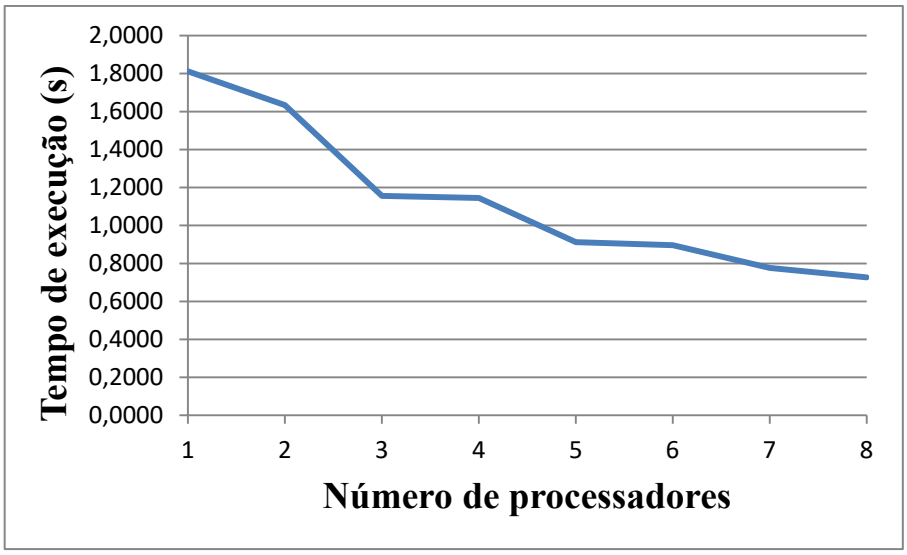



Figura 3.24: Tempo de execução Spark – Tratamento CSV segunda execução

Assim, percebe-se o ganho de desempenho na reutilização dos dados em memória, pois o Spark efetua a leitura do dado e o armazena na memória para ser utilizado posteriormente, melhorando a perfomance do *framework*.

Agora com memória de 1024 MB, o tempo de execução apresentou tempos de execução similares aos tempos com 512 MB. Mesmo quando foi utilizado 16 GB, não apresentou melhora significativa, logo se infere que para esse arquivo o mínimo necessário de memória é 512 MB para atingir o máximo de desempenho.

### 3.4.3 Apache Spark: modelagem da rede com K-means

Os testes de desempenho na utilização do K-means se darão em dois casos: o primeiro utilizará uma captura com grande quantidade de segmentos para treinar o algoritmo, portanto será avaliado o tempo gasto com o aprendizado e modelagem da rede. Enquanto no segundo caso, será utilizada a captura de dez minutos para ser aplicada no modelo do K-means aprendido.

Foram utilizados doze tipos de *clusters*, cada um representando uma característica definida para o modelo, com parâmetros de configuração:

- K = 3; e
- Número de interações = 7.

A figura 3.25 mostra o tempo de execução da aprendizagem do K-means com CSV de 85 MB, que contém 1.083954 milhão de conexões, utilizando 1024 MB de memória RAM com o número de processadores igual a 1, 4 e 8.

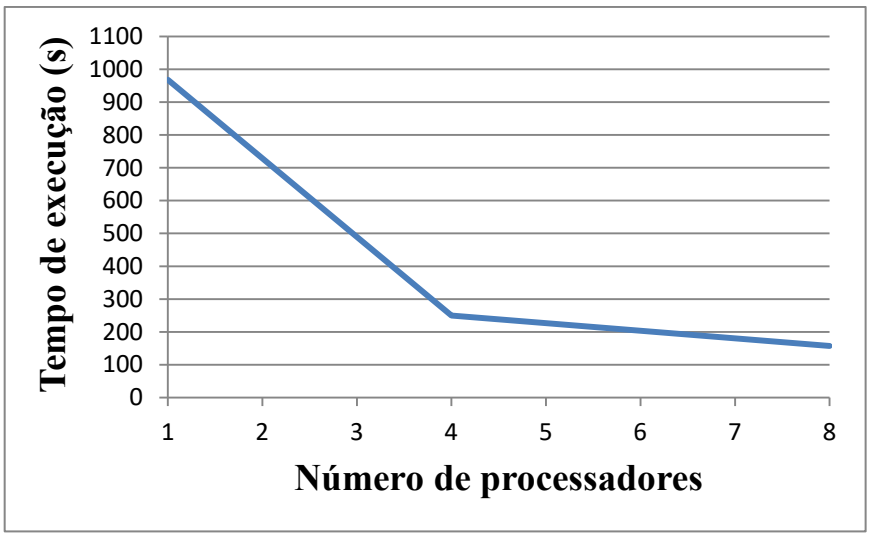



Figura 3.25: Tempo de execução – aprendizagem K-means variando processadores

Percebe-se a influência da quantidade de processadores ao utilizar o Spark, pois o tempo de execução pode ser reduzido em aproximadamente seis vezes quando se utiliza oito núcleos de processamento, em vez de apenas um.

Em seguida, será verificado o impacto da quantidade de memória RAM no tempo de execução, para isso a quantidade de processadores será oito. A quantidade de memória alocada para o Spark será iniciada em 512 MB e os testes seguintes terão o dobro do valor anterior, conforme figura 3.26.

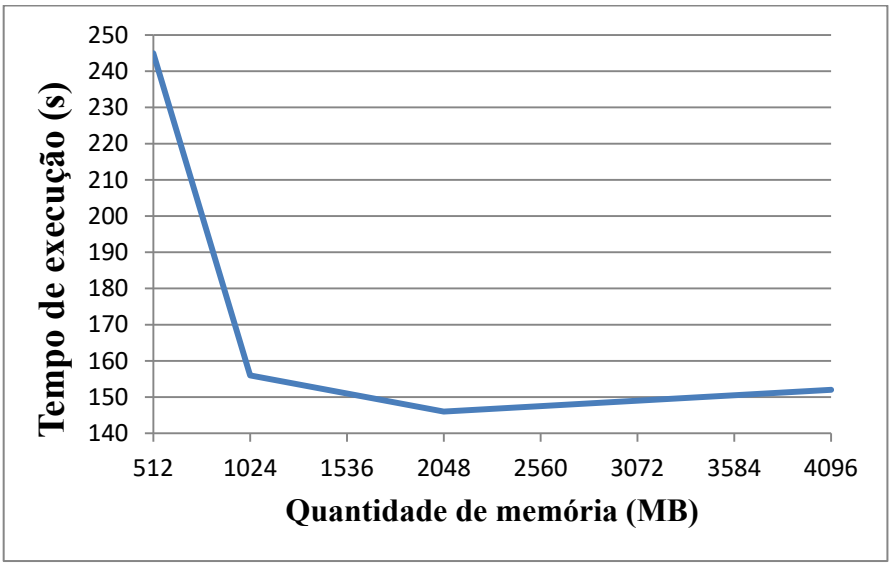



Figura 3.26: Tempo de execução – aprendizagem K-means variando memória

Observa-se que não há muito ganho de desempenho ao aumentar a quantidade de memória a partir de 1024 MB. Logo, para executar a aprendizagem do K-means com arquivo de 85 MB não será necessário alocar memória RAM superior a 1024 MB, pois o desempenho não melhora muito. Além disso, esta fase do modelo não necessita ser inferior a cinco minutos, porque será efetuada apenas uma vez e será usada ao carregar o modelo armazenado.

Agora, iniciando a segunda parte do teste de desempenho do K-means, a partir da criação do modelo da rede com a aprendizagem do algoritmo, será testada a captura com tráfego de dez minutos, a fim de obter o tempo de execução do agrupamento das conexões.

Para isso, deve-se efetuar o carregamento do modelo treinado, e em seguida, executa-se a designação de grupo para cada conexão. Doravante, serão utilizados em todos os testes oito processadores, sendo alterada apenas a quantidade de memória RAM.

Assim, na figura 3.27, pode-se ver o tempo de execução da fase de agrupamento das conexões com o K-means. Observa-se flutuações no tempo de execução com valor médio de 13 segundos. Como o tempo com 512 MB se aproximou da média e não houve ganho significativo ao aumentar a memória, pode-se definir essa quantidade de memória RAM como a mínima necessária para realizar essa fase.

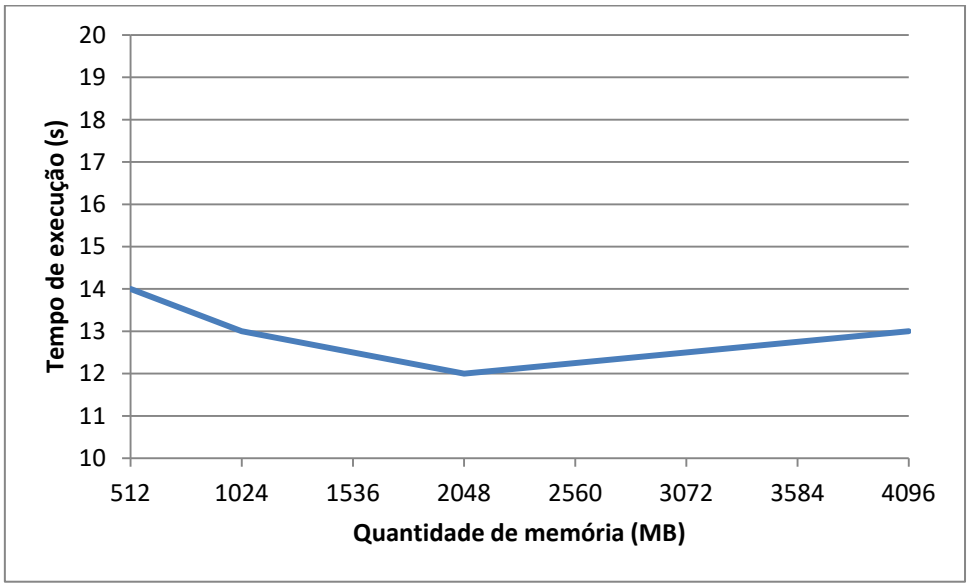



Figura 3.27: Tempo de execução – agrupamento K-means variando memória

Em suma, para executar o agrupamento do K-means com a captura de dez minutos são necessários apenas 512 MB de memória RAM e oito processadores com 2.6 GHZ, concluindo essa fase em 14 segundos.

### 3.4.4 Apache Spark: aprendizagem Random Forest

O próximo passo é testar o tempo gasto para realizar a aprendizagem do *Random Forest*, este estágio executa também a segunda parte da modelagem da rede com K-means, e como suas entradas são o dataframe com o fluxo normal e outro dataframe com o RAT, logo são executadas duas vezes essa parte do K-means.

Portanto, nos testes será mensurado o tempo de execução desde os agrupamentos de cada dataframe até a validação dos dados de treinamento. Para esse teste de desempenho foi utilizado o arquivo de 85 MB, mencionado anteriormente, além de um arquivo com 291 conexões e 404.234 segmentos de RAT, possuindo tamanho de 22.5 KB.

Os parâmetros de configuração do algoritmo *Random Forest* foram:

- Impureza = gini;
- Profundidade das árvores = 16; e
- Número máximo de árvores = 129.

Neste caso, iniciou-se com 1024 MB de memória para o Spark, porém o spark-shell parou de funcionar, porque a memória foi insuficiente. Isso aconteceu novamente ao aumentar para 2048 MB e 4096 MB.

Agora com 8192 MB de memória foi possível executar a aprendizagem do *Random Forest*. Houve flutuações no tempo de execução, de acordo com a figura 3.28, mas o tempo médio ficou em 183.57 segundos, sendo valor próximo ao obtido com a memória alocada de 10 GB com tempo de 184 segundos.

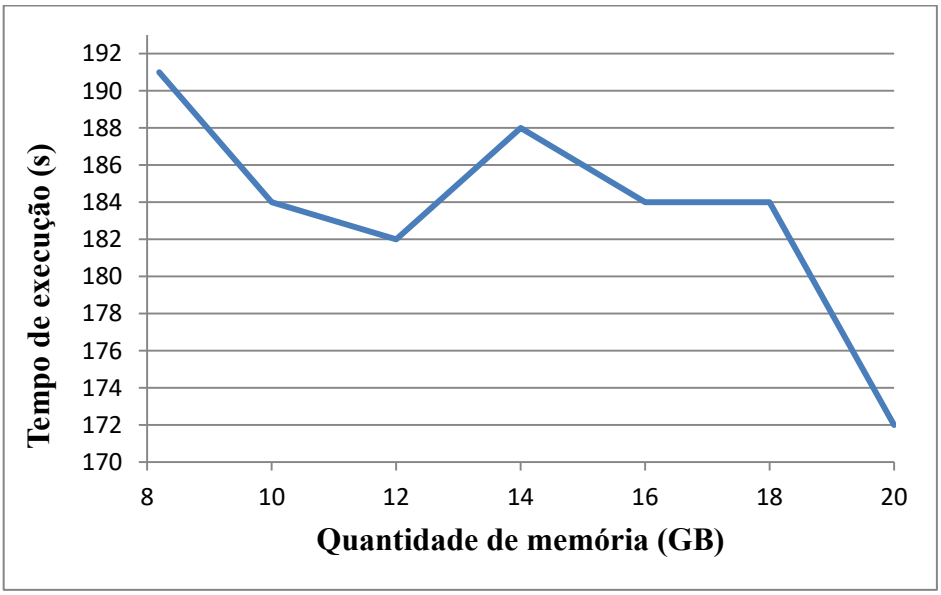



Figura 3.28: Tempo de execução – aprendizagem *Random Forest* variando memória

Assim, a quantidade de memória mínima para esse estágio foi de 10 GB, para oferecer o melhor desempenho ao utilizar um CSV de 85 MB com tráfego normal junto com outro CSV de RAT com 22.5 KB.

### 3.4.5 Apache Spark: classificação dos fluxos com Random Forest

Concluindo os testes de execução do sistema de detecção, será efetuado um teste completo desde a criação dos dataframes até a validação do modelo, excluindo o estágio de extração de características e as fases de aprendizagem dos algoritmos.

Os parâmetros dos algoritmos foram mantidos, mas os arquivos utilizados serão diferentes. Para realizar esse teste, utilizaram-se os seguintes arquivos:

- CSV com fluxos normais - 3255 conexões, 309.521 segmentos e tamanho 265 KB; e
- CSV com fluxos de RAT – 161 conexões, 267.958 segmentos e tamanho 12 KB.

Agora o teste iniciará com a quantidade de memória RAM de 512 MB, sendo adicionadas 1024 MB em cada teste, conforme figura 3.29. Percebe-se que não há ganho significativo ao aumentar a memória, pois considerando o valor médio de 36.5 segundos dos testes, a quantidade de memória mínima atingiu 39 segundos, sendo portanto bem próximo da média.

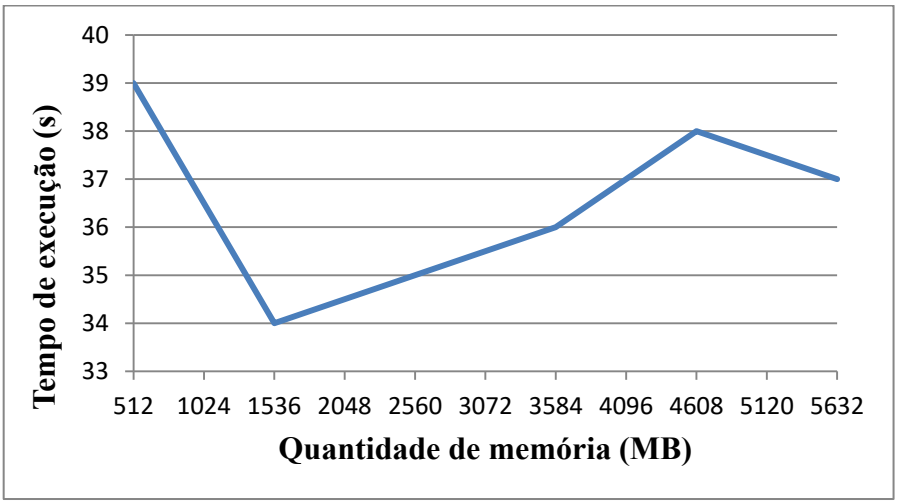



Figura 3.29: Tempo de execução – classificação *Random Forest* variando memória

Cabe ressaltar que em todos os testes foi utilizada uma captura com tempo duas vezes superior ao requisito de cinco minutos, exceto na aprendizagem dos algoritmos. Mesmo assim o tempo de execução dos estágios, após o aprendizado dos algoritmos, no Spark foi de 39 segundos e no Tshark foi de 34 segundos, resultando no tempo total de 73 segundos ou um minuto e 13 segundos, sendo muito inferior ao tempo limite, isso utilizando a configuração mínima de 512 MB de memória RAM e oito processadores de 2.6 GHZ. Por fim, a tabela 3.6, resume os tempos de execução encontrados em cada teste realizado.

Tabela 3.6: Resultados dos testes de execução dos componentes do projeto

| **Teste** | **Tamanho arquivo** | **Memória mínima** | **Tempo de execução** |
|---|---|---|---|
| Extração das características | 28.5 MB | 256 MB | 35 segundos |
| Tratamento das características | 265 KB | 512 MB | 4.3 segundos |
| Modelagem da rede com K-means (aprendizagem) | 85 MB | 1024 MB | 156 segundos |
| Modelagem da rede com K-means (agrupamento) | 265 KB | 512 MB | 14 segundos |
| Aprendizagem *Random Forest* | 85 MB e 22.5 KB | 10 GB | 184 segundos |
| Classificação *Random Forest* | 265 KB e 12 KB | 512 MB | 39 segundos |

# Capítulo 4

# Resultados

Neste capítulo serão apresentados dois estudos de casos para validar o modelo de detecção de RAT. O primeiro caso retrata um ambiente real, sendo o modelo aplicado em um cenário com diversos departamentos de uma empresa. Enquanto o segundo busca reproduzir os resultados do primeiro caso, utilizando capturas de RAT disponíveis na *internet*.

## 4.1 Validação do sistema de detecção

Após a compreensão das características dos RAT e criação dos componentes do projeto, é necessário verificar a eficácia do sistema de detecção. Existem diversas formas de utilizar os algoritmos de *machine learning*, assim como há diversos tipos de algoritmos, e, além disso, podem ser usadas diversas características para representar o comportamento de uma rede, bem como combiná-las para criar novas características.

Escolher quais algoritmos utilizar, a forma de aplicá-los, os valores dos parâmetros dos algoritmos e quais características serão empregados, representa tarefa complexa de se executar.

Uma forma de verificar a correta aplicação dos algoritmos é utilizar o estágio de validação para confirmar a acurácia e outra métricas de desempenho do modelo criado. Caso não esteja conforme ao especificado, devem-se alterar os parâmetros dos algoritmos, características usadas, ou até mesmo, confirmar incapacidade de detecção dos algoritmos empregados, seja pela forma que foram utilizados ou pela ineficiência do algoritmo.

### 4.1.1 Estudo de caso: validação em ambiente real

O cenário analisado para extrair as capturas de tráfego da rede interna, a fim de treinar o modelo de detecção, pode ser visto na figura 4.1, de forma simplificada, a topologia da rede. A máquina virtual do projeto efetua a captura de todo do tráfego de entrada e saída dos dois departamentos, por meio de espelhamento da porta conectada ao roteador. Os departamentos são separados por VLAN distintas e possuem computadores, *access point* e impressoras, sendo todos conectados diretamente ao *switch*.

Os usuários podem acessar a rede de armazenamento de dados, serviços internos, serviços na *internet*, acesso remoto a *switches* e a servidores, etc. Assim, há diversos tipos de conexões realizadas nessa rede interna.

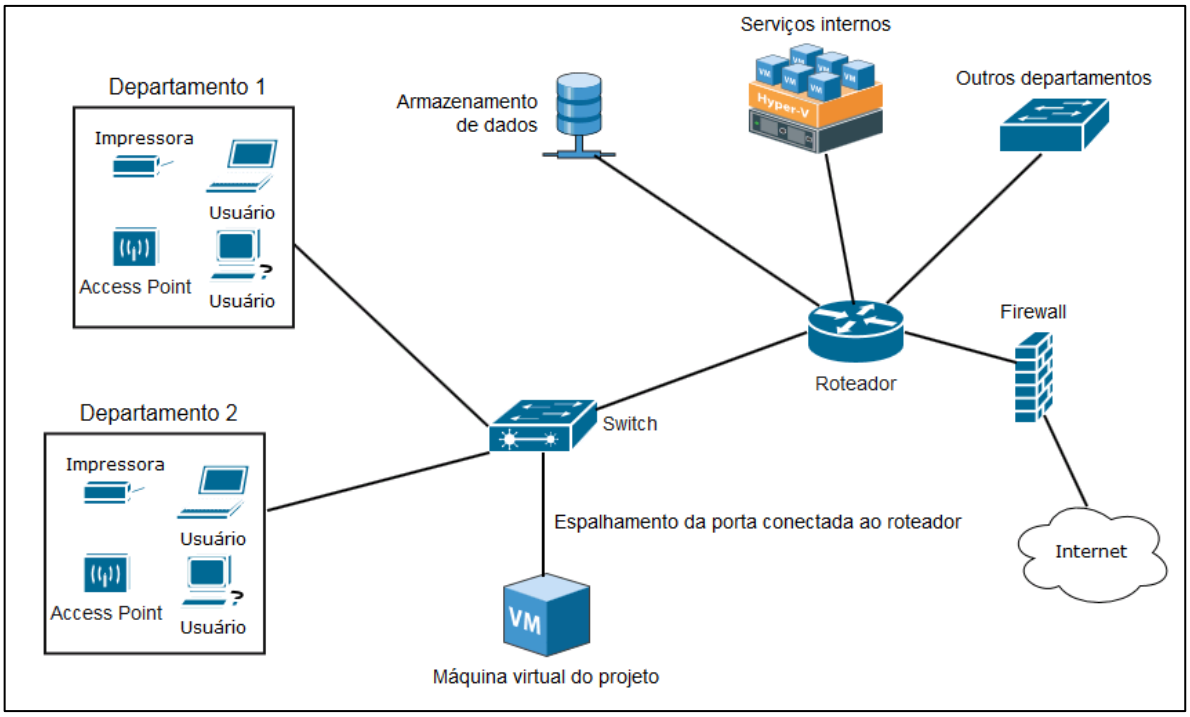



Figura 4.1: Rede interna analisada

A captura usada para modelar o comportamento da rede apresentada foi executada sem interrupções, durante uma semana, capturando inclusive conexões nos finais de semana. Após executar a captura, o arquivo PCAP gerado possuia tamanho original de 179 GB, entretanto, após a redução de 58 bytes por pacote, ficou com 23 GB e aproximadamente 237 milhões de pacotes.

Porém, ao executar o estágio de extração das características, não foi possível concluir a execução, porque a memória da máquina virtual não foi suficiente para processar essa quantidade de pacotes com o Tshark.

Assim, reduziu-se a quantidade de pacotes em 20 milhões em cada teste, sendo que o Tshark só foi capaz de concluir a execução quando restaram 100 milhões de pacotes. Portanto, esse foi o limite de captura com a configuração da máquina virtual do projeto, utilizando o Thark.

Para contornar essa situação, pode-se dividir a captura em partes menores e executá-las individualmente no Tshark, e ao final, juntar todas as estatísticas em um único arquivo para treinar os algoritmos. Contudo, foi utilizado para treinamento o arquivo com 100 milhões de pacotes.

Após ter todas as capturas do tráfego da rede interna e dos RAT, iniciou-se a etapa de ajuste do modelo de detecção. Os dados dos arquivos utilizados para treinar os algoritmos estão na tabela 4.1.

Tabela 4.1: Arquivos para treinamento dos algoritmos

| Tipo de captura | Tamanho CSV | Quantidade de pacotes | Quantidade de conexões |
|---|---|---|---|
| Conexão normal | 85 MB | 100 milhões | 1.083.954 |
| RAT | 22.5 KB | 404.234 | 291 |

Como foram feitas diversas combinações de características e alterações dos parâmetros dos algoritmos, serão apresentados aquelas que forneceram a melhor acurácia, obtendo baixa quantidade de falso negativo e de falsos positivos. Em seguida, serão realizadas algumas variações nos valores dos parâmetros, a fim de ilustrar o impacto do ajuste no resultado.

As características utilizadas estão na tabela 3.1, acrescidas de mais treze características que foram obtidas com a aplicação do K-*means* para cada característica individualmente, ou seja, no aprendizado do *Random Forest* serão utilizadas 26 características, os quais treze são agrupamentos do K-means.

O melhor valor dos parâmetros do algoritmo K-means são:

- K = 3; e
- Número de interações = 7.

Enquanto para o Random Forest:

- Profundidade das árvores = 16; e
- Número de árvores = 129.

O resultado do WSSSE de treinamento do K-means é exibido na tabela 4.2.

Tabela 4.2: Resultado WSSSE - aprendizagem K-means

| Cluster - Características | WSSSE |
|---|---|
| 0 - TS | 0.03884997430468126 |
| 1 - BES | 0.0016187692044137827 |
| 2 - SES | 0.00859549599329969 |
| 3 - BEC | 0.001598418070212484 |
| 4 - SEC | 0.05035521432058816 |
| 5 - SES/SEC | 0.08992892204259716 |
| 6 - BES/SES | 27.459767163112243 |
| 7 - TB | 0.001863055540317787 |
| 8 - BES/BEC | 0.003855114206249195 |
| 9 - BEC/SEC | 6.359464820778393 |
| 10 - TS/Duracao | 0.6619248994699641 |
| 11 - TB/Duracao | 0.025283689068941143 |
| 12 - Duracao | 0.08347632705745826 |

O resultado da *confusion matrix* e das métricas de validação do modelo com essas características e parâmetros, usando 1% dos dados de treinamento, pode ser visto na tabela 4.3.

Tabela 4.3: Resultado das métricas de validação - aprendizagem Random Forest

| **Confusion Matrix** | |
|---|---|
| TN= 10880 | FP = 0 |
| FN= 0 | TP= 6 |
| **Métricas** | |
| Accuracy = 1.0 | |
| Precision | |
| label 0 = 1.0 | label 1 = 1.0 |
| Recall | |
| label 0 = 1.0 | label 1 = 1.0 |
| F1-measure | |
| label 0 = 1.0 | label 1 = 1.0 |

A tabela 4.3 mostra que os parâmetros estão adequados para identificar parte dos dados de treinamento. Conforme são alterados os parâmetros, há a possibilidade de ocorrer erro já nessa validação.

Agora para validar a aprendizagem do *Random Forest*, foram utilizados os arquivos da tabela 4.4.

Tabela 4.4: Arquivos para teste dos algoritmos

| **Tipo de captura** | **Tamanho CSV** | **Quantidade de pacotes** | **Quantidade de conexões** |
|---|---|---|---|
| Conexão normal | 265 KB | 309.521 | 3.225 |
| RAT | 12 KB | 267.958 | 161 |

A tabela 4.5 exibe o resultado da validação ao aplicar o modelo de classificação criado com os parâmetros e características descritos anteriormente.

Tabela 4.5: Resultado das métricas de validação - classificação Random Forest

| **Confusion Matrix** | |
|---|---|
| TN= 3102 | FP = 153 |
| FN= 16 | TP= 146 |
| **Métricas** | |
| Accuracy = 0.9505269320843092 | |
| Precision | |
| label 0 = 0.994868505452213 | label 1 = 0.4865771812080537 |
| Recall | |
| label 0 = 0.9529953917050691 | label 1 = 0.9006211180124224 |
| F1-measure | |
| label 0 = 0.9734818766671897 | label 1 = 0.6318082788671024 |

Observa-se que o modelo de detecção apresenta 95% de acurácia, 16 conexões de RAT classificadas erradamente como normais, representando 9.9% de falsos negativos, e 153 conexões normais foram identificadas como RAT, o que equivale a 4.7% de falsos positivos.

Os clusters que tiveram elevada soma de erros foram com as características BES/SES e BEC/SEC, tabela 4.2. Assim foram feitos testes retirando essas duas características para tentar melhorar a acurácia. Porém, conforme tabela 4.6, o resultado foi pior, porque a taxa de falsos positivos triplicou. Embora tenha reduzido a taxa de falsos negativos, o que é muito importante, deve-se buscar o equilíbrio de redução entre as duas taxas.

Tabela 4.6: Resultado das métricas de validação – sem as características com elevado WSSSE

| **Confusion Matrix** | |
|---|---|
| TN= 2786 | FP = 469 |
| FN= 5 | TP= 152 |
| **Métricas** | |
| Accuracy = 0.8600702576112412 | |
| Precision | |
| label 0 = 0.9967799642218247 | label 1 = 0.24476650563607086 |
| Recall | |
| label 0 = 0.8559139784946237 | label 1 = 0.9440993788819876 |
| F1-measure | |
| label 0 = 0.9209917355371902 | label 1 = 0.38874680306905374 |

Em seguida, foi realizado um teste com todas as características, adicionando uma nova, denominada SES/Duracao, que é a taxa média de envio de segmentos pelo servidor. Contudo, de acordo com a tabela 4.7, a acurácia também teve redução. Isso contraria o estudo de Jiang e Omote (2015), figura 2.21, o qual mostrou que a

acurácia do *Random Forest* é aumentada quando se adicionam características ao modelo. Mas, como neste projeto estão sendo utilizados dois algoritmos, o qual um influencia o resultado do outro, deve-se observar o impacto das características em ambos.

Tabela 4.7: Resultado das métricas de validação – adicionando SES/Duracao

| **Confusion Matrix** | |
|---|---|
| TN= 2985 | FP = 270 |
| FN= 11 | TP= 150 |
| **Métricas** | |
| Accuracy = 0.9177400468384075 | |
| Precision | |
| label 0 = 0.9963284379172229 | label 1 = 0.35714285714285715 |
| Recall | |
| label 0 = 0.9170506912442397 | label 1 = 0.9316770186335404 |
| F1-measure | |
| label 0 = 0.9550471924492081 | label 1 = 0.5163511187607572 |

Portanto foram mantidas as características da tabela 3.1, durante o restante da validação e testes.

A seguir, foram feitas algumas combinações dos parâmetros para ilustrar os seus efeitos. A figura 4.1 mostra a variação da acurácia, taxa de falsos negativos e falsos positivos, quando é alterado apenas o valor máximo de profundidade das árvores.

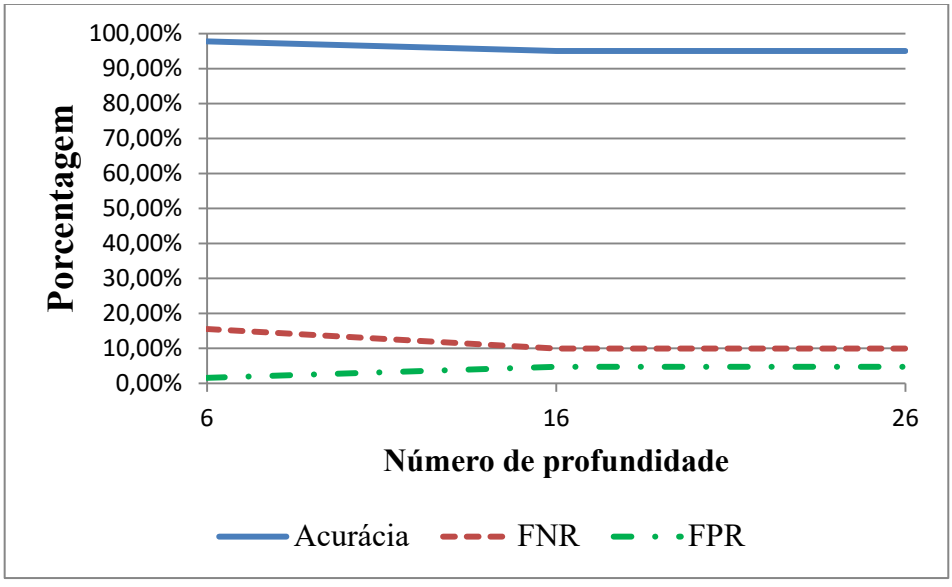



Figura 4.1: Impacto ao alterar o parâmetro: profundidade das árvores no *Random Forest*

Nota-se que quando é usado o valor igual a 16, a acurácia reduziu de 97.8% para 95%, há redução da taxa de falsos negativos (FNR) de 15.5% para 9.9%, e a taxa de falsos positivos (FPR) foi de 1.5% para 4.7%. Observa-se também que a partir do valor 16 não ocorre mais ganho nem perda no modelo.

O próximo parâmetro a ser avaliado será o número máximo de árvores usadas pelo *Random Forest*, a figura 4.2 exibe o resultado da variação.

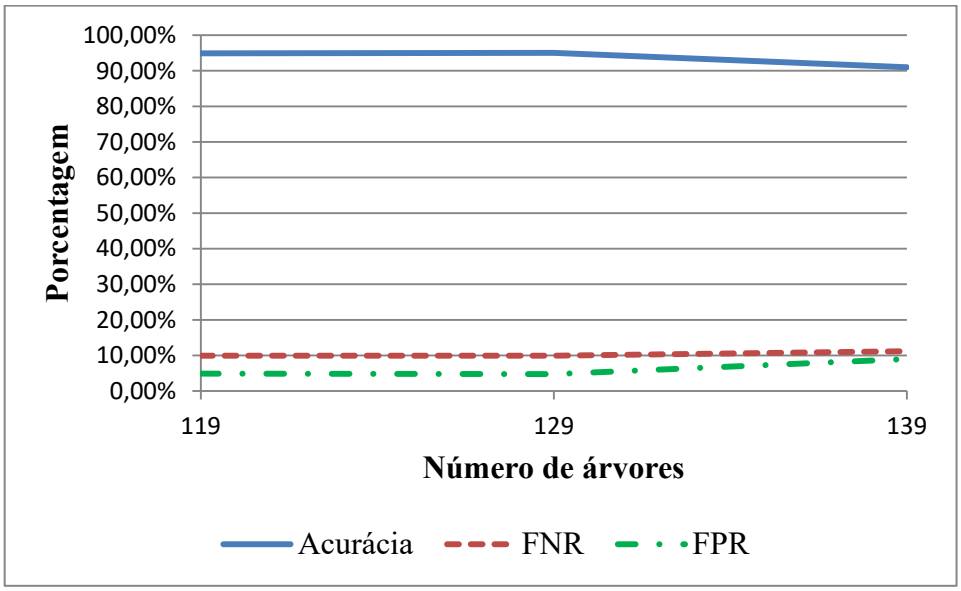



Figura 4.2: Impacto ao alterar o parâmetro: número de árvores no *Random Forest*

A acurácia neste caso variou de 94.9% para 95% quando foi alterado o valor de 119 para 129, a FNR se manteve constante em 9.9% e a FPR reduziu de 4.9% para 4.7%. Enquanto com o valor de 139, a acurácia caiu para 90.9%, a FNR foi para 11.1% e a FPR aumentou para 9%.

Continuando com a verificação dos valores dos parâmetros, agora a figura 4.3 exibe a variação quando é alterado o valor do K no K-*means*.

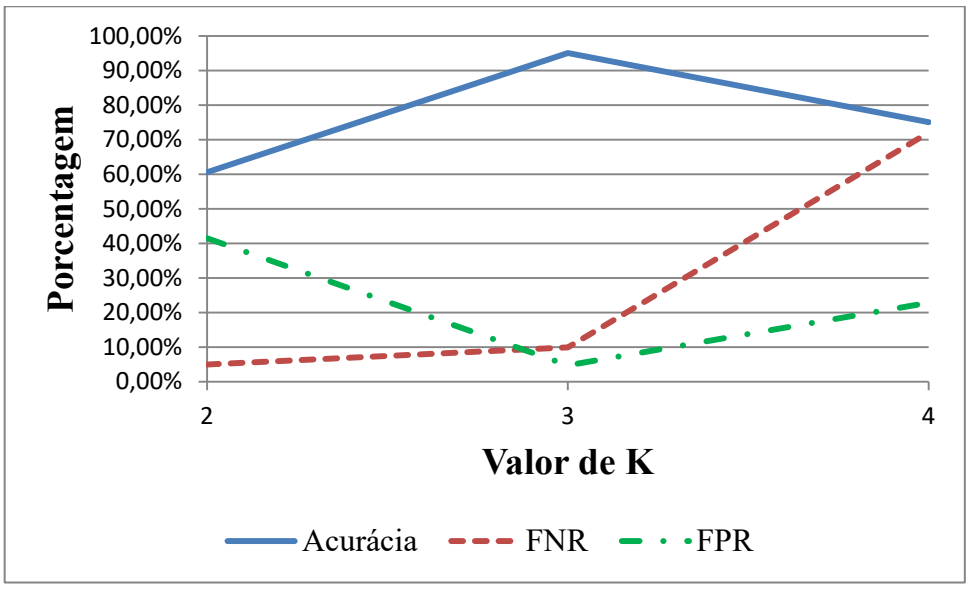



Figura 4.3: Impacto ao alterar o parâmetro: K no K-*means*

Neste caso, os valores tiveram grande impacto no resultado final do modelo. A acurácia iniciou com 60.5% com K igual a dois, passando para 95% com K igual a três, e terminou em 75% com K igual a quatro. Isso mostra a importância de se ajustar os valores dos parâmetros envolvidos.

Durante a evolução do valor de K, a FNR sai de 4.9% para 72%. Enquanto a FPR oscilou de 41% para 4.7%, e depois foi para 22.8%.

O ajuste nesse algoritmo se tornou difícil de compreender por que, de acordo com a literatura, quando se aumenta o valor de K o desempenho é melhorado, por isso, inicialmente foi definido utilizar um valor elevado e que tenha correlação com o projeto. Assim o valor escolhido foi calculado a partir da quantidade de portas de servidor distintas nas conexões.

Por exemplo, na captura de treinamento há 2547 valores distintos de portas utilizadas, logo essa quantidade foi usada para configurar o K do K-*means*, o que resultou em acuraria de 90.3%, FNR de 36.6% e FPR de 8.3%. Assim, se for comparar somente a acurácia, o acréscimo em relação a K igual a quatro foi realmente superior, entretanto quando se trata de sistema de detecção de intrusão, a métrica mais importante é a FNR. Portanto, após diversos valores de K testados, a fim de reduzir a FNR, encontrou-se o valor K igual a três como melhor para esse tipo de aplicação do algoritmo.

Por fim, o último parâmetro testado foi o número de interações realizado pelo K-*means*, cujo objetivo é ajustar o agrupamento das conexões. A figura 4.4 exibe o resultado desse teste.

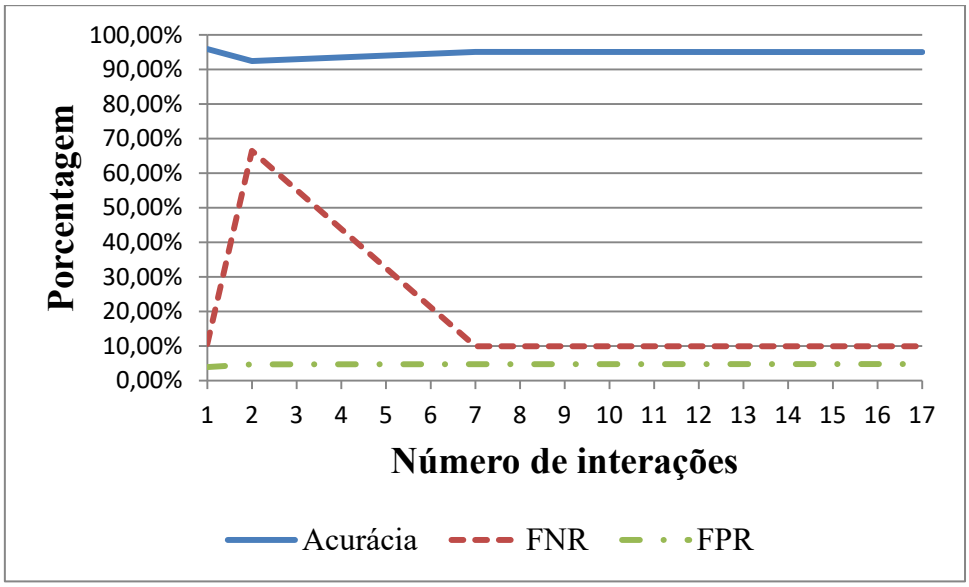



Figura 4.4: Impacto ao alterar o parâmetro: número de interações no K-*means*

Com apenas uma interação, o modelo apresentou 95.9% de acurácia, FNR de 10.5% e FPR de 3.9%. Quando o número de interações foi igual a dois, houve queda de desempenho em todas as métricas, ficando em 92.4% a acurácia, 66.4% de FNR e 4.7% de FPR. Entretanto com valor igual sete, o resultado foi 95% de acurácia, 9.9% de FNR e 4.7% de FPR, mesmo quando o valor foi aumentado para 17, essas taxas se mantiveram constante.

### 4.1.2 Estudo de caso: aplicação do modelo em capturas de RAT de terceiros

Como o projeto tem o objetivo de criar um sistema de detecção em tempo real capaz de detectar qualquer tipo de RAT, além da validação com capturas criadas para o desenvolvimento do modelo, serão utilizadas capturas que não foram padronizadas com a metodologia de captura adotada.

Para isso foi pesquisado na *internet* dois tipos de capturas de RAT distintos e que não fazem parte dos RAT de treinamento do algoritmo. Também será testada a captura usada por Pallaprolu et al. (2016), denominada de dataset 1, a fim de comparar os resultados deles com os deste projeto.

Destaca-se que nos testes desta seção, todo o tráfego das capturas de terceiros são consideradas como RAT, fato que foi confirmado ao analisar o conteúdo das capturas, pois as portas dos servidores possuiam o comportamento de incrementar o seu valor. Além disso, alguns autores das capturas informam que há somente pacotes referentes à RAT.

A validação se dará em duas etapas, a primeira usará os mesmos cinco RAT de treinamento para treinar os algoritmos, enquanto na segunda, serão utilizadas todas as capturas criadas de RAT no treinamento dos algoritmos, ou seja, tanto os RAT de treinamento quanto os de teste irão ser empregados para o aprendizado dos algoritmos, totalizando dez tipos de RAT para treinamento.

A tabela 4.8 contém as informações das capturas obtidas de terceiros.

Tabela 4.8: Dados das capturas de terceiros

| Tipo de RAT | Porta cliente | Tamanho CSV | Quantidade de pacotes | Quantidade de conexões | Fonte |
|---|---|---|---|---|---|
| GHOST | 4433 | 845 Bytes | 779 | 8 | Sanders (2016) |
| NJRAT | 5552 | 12.1 KB | 52.763 | 147 | Stratosphere IPS project (2015) |
| RAMNIT | 80 e 443 | 28.1 KB | 2.256 | 369 | Computer security (2016) |

Os resultados dos testes da primeira etapa podem ser visualizados na tabela 4.9. Foi adicionado o resultado da seção 4.1.1 para facilitar a análise entre os resultados. Assim como foi feito anteriormente, as métricas apresentadas serão acurácia, FNR e FPR. O tipo de RAT denominado LAB, significa que são os cinco tipos de RAT usados como teste durante o projeto, conforme tabela 3.2.

Tabela 4.9: Comparativo dos resultados das capturas de terceiros – 5 RAT treino

| Tipo de RAT | Acurácia | FNR | FPR |
|---|---|---|---|
| LAB | 95% | 9.9% | 4.7% |
| GHOST | 95.2% | 12.5% | 4.7% |
| NJRAT | 91.7% | 86.3% | 4.7% |
| RAMNIT | 86.8% | 88% | 4.7% |

Ao usar apenas cinco RAT de treino, o modelo proposto foi capaz de classificar corretamente o RAT Ghost com FNR de 12.5%, atingindo taxa próxima do melhor resultado do projeto, que foi de 9.9%.

Contudo, quando o modelo é aplicado aos outros dois RAT, tem-se FNR superior a 86%, o que mostra ineficiência de detecção do modelo. A FPR se manteve constante em 4.7%, porque o tráfego considerado normal não foi alterado.

Agora na segunda etapa da validação, a tabela 4.10 exibe o resultado quando é aplicado dez RAT para o treinamento do modelo.

Tabela 4.10: Comparativo dos resultados das capturas de terceiros – 10 RAT treino

| Tipo de RAT | Acurácia | FNR | FPR |
|---|---|---|---|
| LAB | 85.6% | 2.4% | 15.1% |
| GHOST | 84.9% | 37.5% | 15.1% |
| NJRAT | 84.2% | 34% | 15.1% |
| RAMNIT | 82.1% | 43.3% | 15.1% |

Nesta etapa, percebe-se redução na acurácia do modelo, isso ocorreu por que a taxa de falsos positivos triplicou. Nota-se que ao acrescentar o dobro de RAT no treinamento dos algoritmos, foi possível reduzir a FNR, em aproximadamente 50%, quando foi aplicado o NjRAT e Ramnit, porém isso aumentou FNR do Ghost.

Infere-se que aumentando a quantidade de RAT para treino, além de ajudar classificar o comportamento de outros RAT, amplia-se a chance de ocorrer comportamentos similares entre conexões normais e RAT, o que resulta em maior erro de classificação dos algoritmos.

E no caso do LAB, a FNR foi reduzida em 75%, pois todo o seu tráfego foi empregado no treinamento, assim o algoritmo aprendeu previamente todo o seu comportamento, o que facilitou a classificação.

Entretanto essa redução deveria ser de 100%, pois a mesma captura foi apresentada no treinamento e no teste. Assim, esse resultado divergente evidencia que o modelo ainda precisa de ajustes e melhorias.

A captura do RAT Ghost foi a mesma utilizada no trabalho do Pallaprolu et al. (2016), e o seu resultado está na figura 2.26. Embora a implementação e as

características foram diferentes deste projeto, a tabela 4.11 compara a capacidade de detecção dos dois estudos, seguindo as métricas utilizadas por esse autor.

A divisão do RAT Ghost em *small*, *medium* e *large* foi feita para adequar o resultado do trabalho relacionado, pois foi dessa forma que Pallaprolu et al. (2016) executaram a sua pesquisa. E o resultado GHOST - LAB é o deste projeto utilizando a captura Ghost com cinco RAT de treinamento, conforme tabela 4.9.

Tabela 4.11: Comparativo dos resultados de um trabalho relacionado e o projeto

| **Tipo de RAT** | **Precision** | **Recall** | **F measure** |
|---|---|---|---|
| GHOST - LAB | 4.3% | 87.5% | 8.3% |
| GHOST - *small* | 100% | 100% | 100% |
| GHOST – *medium* | 70% | 100% | 85% |
| GHOST – *large* | 68% | 80% | 70% |

Não foi possível comparar corretamente o *precision* e o F *measure*, porque seus resultados dependem da quantidade de falsos positivos, e o estudo elaborado por esses autores considerou que apenas uma parte da captura tinha conexão de RAT, o que não é verdade, pois todos os pacotes fazem parte da comunicação do RAT.

Dessa forma, dentre os 779 pacotes, 9 foram considerados como ataque e 770 como normais. Ainda foi utilizado o mesmo RAT para treinamento e teste do modelo, logo influenciou o resultado.

Eles usaram a assinatura ANABILGI do cabeçalho do RAT para identificar os pacotes do *malware*, assim os pacotes que não tinha esse cabeçalho foram classificados como normais.

Como este projeto pretende identificar apenas comportamentos das conexões, foi considerada toda a captura como ataque, além disso, não foi usado o mesmo RAT para treinamento e teste. E o tráfego considerado como normal foi completamente distinto do considerado anormal, esta parte do teste do modelo justifica a baixa taxa de *precision* e F *measure*, porque a quantidade de pacotes e conexões normais utilizadas foi muito superior as de RAT, o que aumenta a relação da taxa de falsos positivos e a taxa de falsos negativos. Assim, essas duas métricas não podem ser comparadas entre os dois estudos.

Como a métrica *recall* avalia somente a quantidade de acertos e erros da classificação do RAT ou conexão normal. Considerando o recall do RAT, percebe-se que o resultado do modelo proposto neste projeto foi superior ao estudo elaborado por Pallaprolu et al. (2016) , quando foi utilizado o GHOST – large.

Isso indica maior capacidade de detecção deste projeto, pois não se limita a um tipo de RAT e nem a verificação de assinaturas de ataques, e ainda pode oferecer maior *recall*, dependendo do caso.

# Capítulo 5

# Conclusão

Neste trabalho, foi proposto e desenvolvido um sistema de detecção de RAT com alta acurácia e baixo custo computacional, baseado em anomalias de comportamento de rede, que utiliza a combinação de dois algoritmos de *machine learning*, *K-means e Random Forest*, com aplicação de características de comportamento estudadas em trabalhos relacionados. Assim, neste capítulo serão apresentadas as contribuições deste projeto e as melhorias que deverão ser feitas em trabalhos futuros.

O sistema de detecção proposto se mostra promissor na classificação de conexões de RAT, sem necessitar de aprendizagem prévia do malware identificado, pois é capaz de distinguir as conexões somente com o comportamento similar existente entre essa classe de malware.

Ao comparar os resultados publicados, em pesquisas recentes que utilizam essa abordagem de detecção, tabela 5.1, percebe-se que o modelo proposto apresenta resultado satisfatório, mas que ainda precisa de ajustes e melhorias.

Tabela 5.1: Resultados de trabalhos similares

| **Autor** | **Acurácia** | **FNR** | **FPR** | **Algoritmos de *machine learning*** |
|---|---|---|---|---|
| Li et al. (2012) | 91% | Não foi divulgado | 3.2% | K-*means* |
| Jiang e Omote (2015) | 97.1% | 10% | 2.3% | *Random Forest* |
| Modelo proposto | 95% | 9.9% | 4.7% | K-*means* e *Random Forest* |

A escolha das características foi baseada nos principais trabalhos relacionados encontrados, Li et al. (2012) e Jiang e Omote (2015), devido à clareza da metodologia empregada e à contribuição apresentada.

Como ocorreu no trabalho de Li et al. (2012), a captura de conexões normais podem influenciar na acurácia do modelo, porque o comportamento das conexões variam bastante, e quando são similares aos RAT, aumenta-se a quantidade de FNR e FPR, reduzindo portanto a acurácia. A fim de melhorar a acurácia esses autores aumentaram o valor de K para 100, representando o seu melhor resultado, conforme tabela 5.1. Entretanto neste projeto, o melhor valor de K foi três. Isso representa ganho de desempenho se comparado com o trabalho de Li et al. (2012), pois quanto maior o valor de K, maior o custo computacional envolvido.

Agora no estudo de Jiang e Omote (2015), não foi possível identificar os valores dos parâmetros utilizados no *Random Forest*, porque utilizaram o método K-Fold

*cross validation* para encontrar os valores dos parâmetros, e não publicaram o resultado dessa avaliação. Então neste caso, não há como comparar esses valores com os utilizados neste projeto.

As características selecionadas possuem grande relevância no resultado deste projeto, pois mesmo quando foram utilizadas as indicadas por esses estudos, apresentou, inicialmente, resultado inferior ao mostrado na tabela 5.1. Portanto, cabe um estudo detalhado do impacto de cada característica, de suas combinações e dos agrupamentos em *clusters*, bem como utilizar métodos de extração dos parâmetros ótimos dos algoritmos, tais como o K-Fold *cross validation*, a fim de melhorar o modelo.

## 5.1 Contribuição

Além de replicar em parte o resultado dos trabalhos relacionados, este estudo apresentou uma nova forma de classificar os fluxos de conexões com algoritmos de *machine learning*. Definiu uma maneira de reduzir o custo computacional da extração e processamento das características de capturas de pacotes, pois com a redução do tamanho dos pacotes, há diminuição do tamanho da captura em 87%. Apresentou a ferramenta Tshark como opção viável de ser utilizada em conjunto com ferramentas de *big data*. Comprovou com o Tshark que ao utilizar arquivos no formato CSV, o tamanho de arquivo gerado é onze vezes menor do que o formato JSON.

Mostrou de forma simplificada, o impacto dos parâmetros de configuração dos algoritmos de *machine learning* utilizados, bem como, superficialmente, o impacto das características de comportamento escolhidas. Especificou os requisitos mínimos de *hardware* para implementação do projeto. Propôs um sistema de detecção de RAT em tempo real, capaz de ser executado em aproximadamente em um minuto. E por fim, oferece o código fonte usado como forma de auxiliar o desenvolvimento de aplicações no *framework* Spark.

## 5.2 Trabalhos Futuros

Tem-se como melhorias a serem aplicadas neste projeto:

- Elaborar um estudo da influência das características das conexões no modelo proposto, com intuito de selecionar as mais adequadas para melhorar a acurácia;
- Realizar estudo aprofundado dos valores ideais dos parâmetros dos algoritmos de *machine learning*;
- Desenvolver o armazenamento das conexões classificadas, em forma de log, para exibição em um *dashboard web*;
- Desenvolver o estágio de verificação de ocorrência de RAT, a fim de armazenar a captura infectada;
- Aprimorar o código que calcula as características extraídas do CSV campos de pacotes, a fim de reduzir o tempo de execução, o que tornaria viável o uso dessa parte do projeto;
- Desenvolver *script* para executar o Tshark com multiprocessamento, pois é uma limitação da ferramenta; e
- Aumentar a quantidade de RAT de treino e teste, a fim de generalizar a aplicação do modelo.

# REFERÊNCIAS BIBLIOGRÁFICAS


ACODEMY. ***Spark: Learn Spark In A DAY! - The Ultimate Crash Course to Learning the Basics of Spark In No Time*** . 1. ed. Acodemy, 2015. p. 137.

AVEN, J. **Apache Spark in 24 Hours, Sams Teach Yourself**. 1. ed. Indiana: Sams Publishing, 2016. p. 592.

BENZMÜLLER, R. ***Malware numbers of the first half of* 2017**. 2017. Disponível em: <https://www.gdatasoftware.com/blog/2017/07/29905-malware-zahlen-des-ersten-halbjahrs-2017>. Acesso em: 24 jun. 2017

BIJONE, M. *A Survey on Secure Network: Intrusion Detection & Prevention Approaches*. ***American Journal of Information Systems***, 2016, Vol. 4, No. 3, p. 69-88. Disponível em: <http://pubs.sciepub.com/ajis/4/3/2/index.html> Acesso em: 24 jun. 2017

BULLOCK, J.; PARKER, J. T. ***Wireshark for Security Professionals: Using Wireshark and the Metasploit Framework***. 1. ed. John Wiley & Sons, 2017. p. 288.

CERT BR. **Cartilha de Segurança para Internet Versão 4.0**. 2012. Disponível em: <https://cartilha.cert.br/livro/cartilha-seguranca-internet.pdf >. Acesso em: 24 jun. 2017

COMPUTER SECURITY. **RAMNIT *Malware* RAT *Remote Access* Trojan Backdoor *Traffic Sample Download* PCAP**. 2016. Disponível em: <http://www.computersecurity.org/malware-traffic-samples/rat-remote-access-trojan/ramnit-malware-rat-remote-access-trojan-backdoor-traffic-sample-download-pcap/> Acesso em: 24 jun. 2017

CONNECT TROJAN. **Repositório de *malwares*.** 2017. Disponível em: <http://www.connect-trojan.net/> Acesso em: 24 jun. 2017

DUVVURI , S.; SINGHAL, B. **Spark *for Data Science***. 1. ed. Birmingham: Packt Publishing Ltd, 2016. p. 344.

ERMAN, J. et al. *Traffic Classification Using Clustering Algorithms*. ***In Proceedings of the* 2006 SIGCOMM *workshop on Mining network data* (MineNet '06)**. ACM, New York, NY, USA, p. 281-286. Disponível em: <https://dl.acm.org/citation.cfm?id=1162679> Acesso em: 24 jun. 2017

FOROUZAN, B. A. **Comunicação de Dados e Redes de Computadores**. 4. ed. São Paulo: Mc Graw Hill, 2007. p. 1134.

FRAMPTON, M. **Mastering Apache Spark**. 1. ed. Birmingham: Packt Publishing, 2015. p. 320.

GATES, M. ***Machine Learning: For Beginners - Definitive Guide for Neural Networks, Algorithms, Random Forests and Decision Trees Made Simple***. 1. ed. Createspace Independent Publishing Platform, 2017. p. 104.

HARTSHORN, S. ***Machine Learning With Random Forests And Decision Trees: A Visual Guide For Beginners***. 1. ed. Amazon Servicos de Varejo do Brasil Ltda, 2016. p. 74.

HUGUENARD LAB. ***Error Sum of Squares* (SSE).** 2017. Disponível em: < https://hlab.stanford.edu/brian/error_sum_of_squares.html/> Acesso em: 24 jun. 2017

INTERNET CRIME COMPLAINT CENTER (IC3). ***The* 2016 *Internet crime Report highlights***. 2017. Disponível em: <https://pdf.ic3.gov/2016_IC3Report.pdf >. Acesso em: 24 jun. 2017

JIANG, D.; OMOTE, K. *An Approach to Detect Remote Access Trojan in the Early Stage of Communication*. **2015 IEEE *29th International Conference on Advanced Information Networking and Applications***, Gwangiu, p. 706-713, 2015. Disponível em: <http://ieeexplore.ieee.org/document/7098042> Acesso em: 24 jun. 2017

KONDALWAR, M. D.; SHELKE, C. J. *Remote Administrative Trojan/Tool (RAT)*. ***International Journal of Computer Science and Mobile Computing***, Vol.3 Issue.3, March-2014, p. 482-487. Disponível em: <http://ijcsmc.com/docs/papers/March2014/V3I3201499a33.pdf>. Acesso em: 24 jun. 2017

KUROSE, J. F; ROSS, K. W. **Redes de computadores e a Internet: uma abordagem top-down**. 6. ed. São Paulo: Pearson Education do Brasil, 2013. p. 656.

LI, S. et al. *A General Framework of Trojan Communication Detection Based on Network Traces*. **2012 IEEE *Seventh International Conference on Networking, Architecture, and Storage***, Xiamen, Fujian, 2012, p. 49-58. Disponível em: <http://ieeexplore.ieee.org/document/6310875/>. Acesso em: 24 jun. 2017

NICOLAS, P. R. **Scala *for Machine Learning***. 1. ed. Birmingham: Packt Publishing Ltd, 2014. p. 420.

NIXCRAFT. **Linux Hotplug a CPU and Disable CPU Cores At Run Time.** 2009. Disponível em: < https://www.cyberciti.biz/faq/debian-rhel-centos-redhat-suse-hotplug-cpu/> Acesso em: 24 jun. 2017

ODERSKY, M. et al. ***Programming in* Scala: *A Comprehensive Step-by-Step Guide***. 3. ed. California: Artima Press, 2016. p. 837.

PALLAPROLU, S. C. et al. *Label Propagation in Big Data to Detect Remote Access Trojans*. **2016 IEEE International Conference on Big Data**, Washington, DC, 2016, p. 3539-3547. Disponível em: <http://ieeexplore.ieee.org/document/7841017> Acesso em: 24 jun. 2017

RSA RESEARCH. ***Peering into Glassrat: A Zero Detection Trojan from China.*** 2015. Disponível em: < https://paper.seebug.org/papers/APT/APT_CyberCriminal_Campagin/2015/2015.11.23.PEERING_INTO_GLASSRAT/GlassRAT-final.pdf>. Acesso em: 24 jun. 2017

RUITER, A. ***Performance measures in Azure ML: Accuracy, Precision, Recall and F1 Score***. 2015. Disponível em: <https://blogs.msdn.microsoft.com/andreasderuiter/2015/02/09/performance-measures-in-azure-ml-accuracy-precision-recall-and-f1-score/>. Acesso em: 24 jun. 2017

SANDERS, C. ***Packets***. 2016. Disponível em: <http://chrissanders.org/packet-captures/> Acesso em: 24 jun. 2017

STRATOSPHERE IPS PROJECT. ***Datasets***. 2015. Disponível em: <https://stratosphereips.org/category/dataset.html> Acesso em: 24 jun. 2017

SULLIVAN, W. ***Machine Learning For Beginners Guide Algorithms: Supervised & Unsupervsied Learning. Decision Tree & Random Forest Introduction***. 1. ed. Amazon Servicos de Varejo do Brasil Ltda, 2017. p. 268.

VAIDYA, T. 2001-2013: *Survey and Analysis of Major Cyberattacks*. ***Computing Research Repository***, v. abs/1507.06673, 2015. Disponível em: <https://arxiv.org/abs/1507.06673>. Acesso em: 24 jun. 2017

VERIZON ENTERPRISE SOLUTIONS. **2017 *Data Breach Investigations Report* 10th *Edition***. 2017. Disponível em: <http://www.verizonenterprise.com/verizon-insights-lab/dbir/2017>. Acesso em: 24 jun. 2017

WIKI ARCHLINUX. **Cgroups.** 2017. Disponível em: < https://wiki.archlinux.org/index.php/cgroups/> Acesso em: 24 jun. 2017

WIRESHARK. **Wireshark *Manual Pages***. 2017. Disponível em: <https://www.wireshark.org/docs/man-pages/>. Acesso em: 24 jun. 2017

WU, S. et al. *Detecting Remote Access Trojans through External Control at Area Network Borders*. **2017 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), Beijing, 2017, p. 131-141.** Disponível em: <http://ieeexplore.ieee.org/document/7966912/> Acesso em: 24 jun. 2017

YAMADA, S. et al. RAT-*based Malicious Activities Detection on Enterprise Internal Networks*. **2015 10th *International Conference for Internet Technology and Secured Transactions* (ICITST)**, London, 2015, p. 321-325. Disponível em: <http://ieeexplore.ieee.org/document/7412113/> Acesso em: 24 jun. 2017

YU, L. et al. *An Unknown Trojan Detection Method Based on Software Network Behavior*. **Wuhan *University Journal of Natural Sciences***, v. 18, Issue 5, p. 369–376, 2013. Disponível em: <https://link.springer.com/article/10.1007/s11859-013-0944-6> Acesso em: 24 jun. 2017

# APÊNDICE A – Exemplo de Ataque do RAT Poison Ivy 2.3.2

## A.1 Instalação do servidor

Neste procedimento será adotada a instalação do servidor na máquina da vítima de forma manual, entretanto geralmente essa instalação é efetuada por meio de *scripts* maliciosos, sem o acesso direto ao computador.

A figura A.1.1 apresenta o arquivo comprimido em formato RAR, que contém o RAT poison ivy versão 2.3.2, baixado em Connect trojan (2017). Junto com o executável, há um manual explicando algumas características e funcionalidades do RAT.

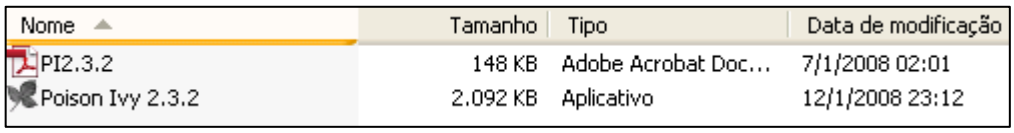



Figura A.1.1: Arquivos baixados

Ao executar o aplicativo Poison Ivy 2.3.2, é exibido, figura A.1.2, um termo de responsabilização pelo uso indevido deste *software*.

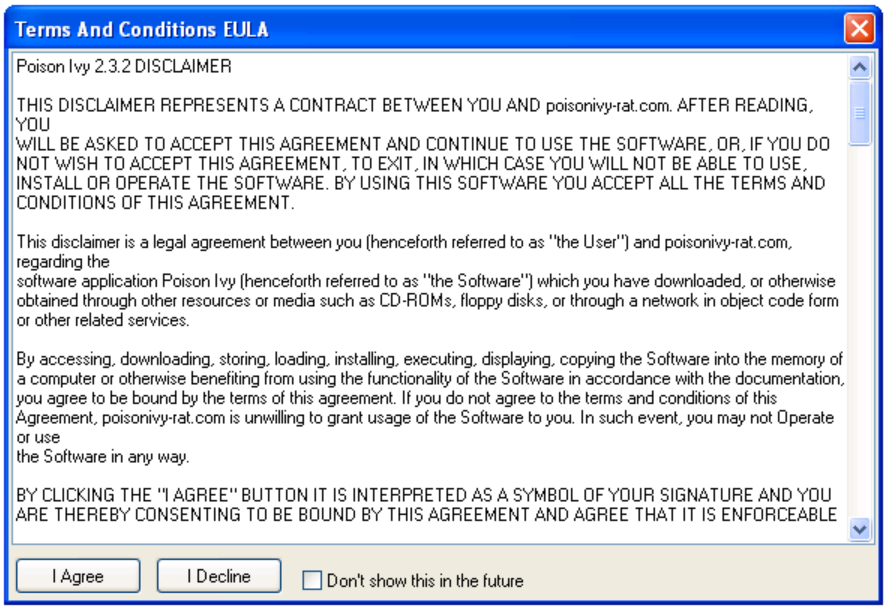



Figura A.1.2: Termo de uso

Na figura A.1.3 é vista a tela inicial do Poison Ivy, percebe-se que na barra status há informação do número de portas abertas, *plugins* ativos e conexões ativas do servidor.

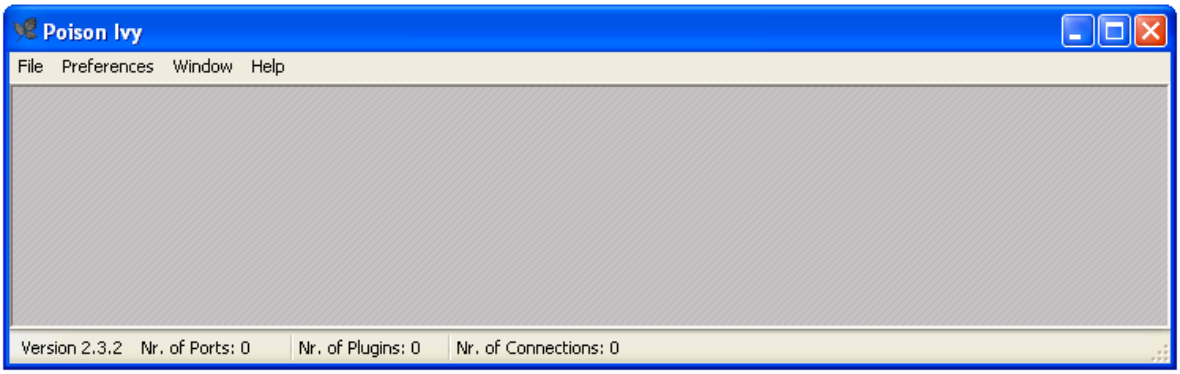



Figura A.1.3: Tela inicial do Poison Ivy

Para criar o servidor, é necessário acessar o “menu File”, e clicar na opção New Server, conforme a figura A.1.4.

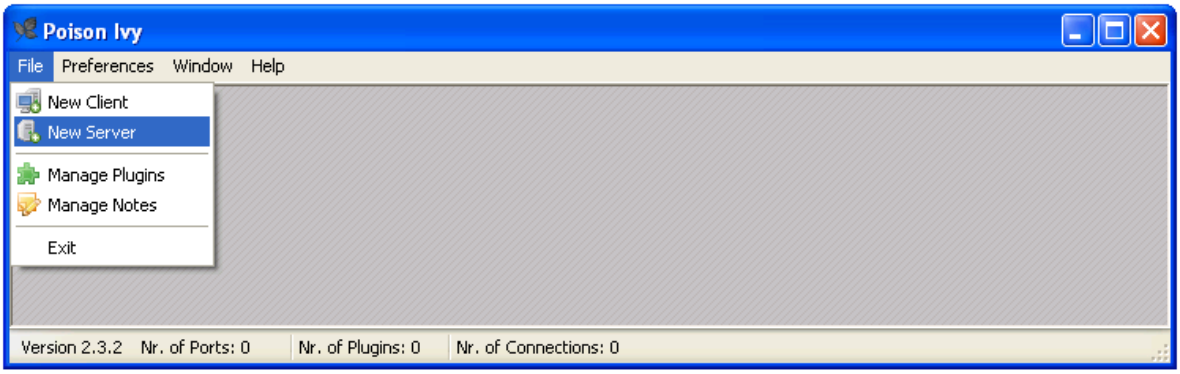



Figura A.1.4: Criar servidor

Em seguida, deve-se criar um perfil de configuração, figura A.1.5, para o servidor a ser criado.

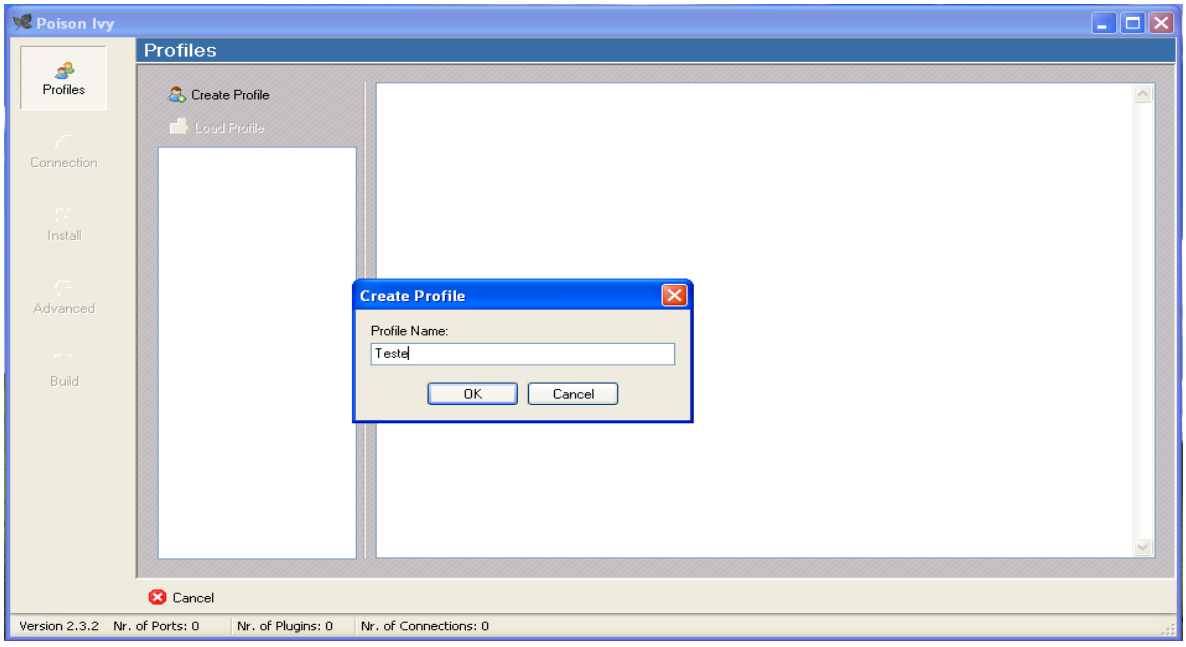



Figura A.1.5: Criar perfil

A seguir, abrirá nova janela para definir a configuração do servidor, figura A.1.6, dentre as opções, tem-se alterar o IP e porta do Servidor, conectar o servidor por meio de proxy, utilizar proxy hijack, definir login e senha ou utilizar arquivo para autenticação.

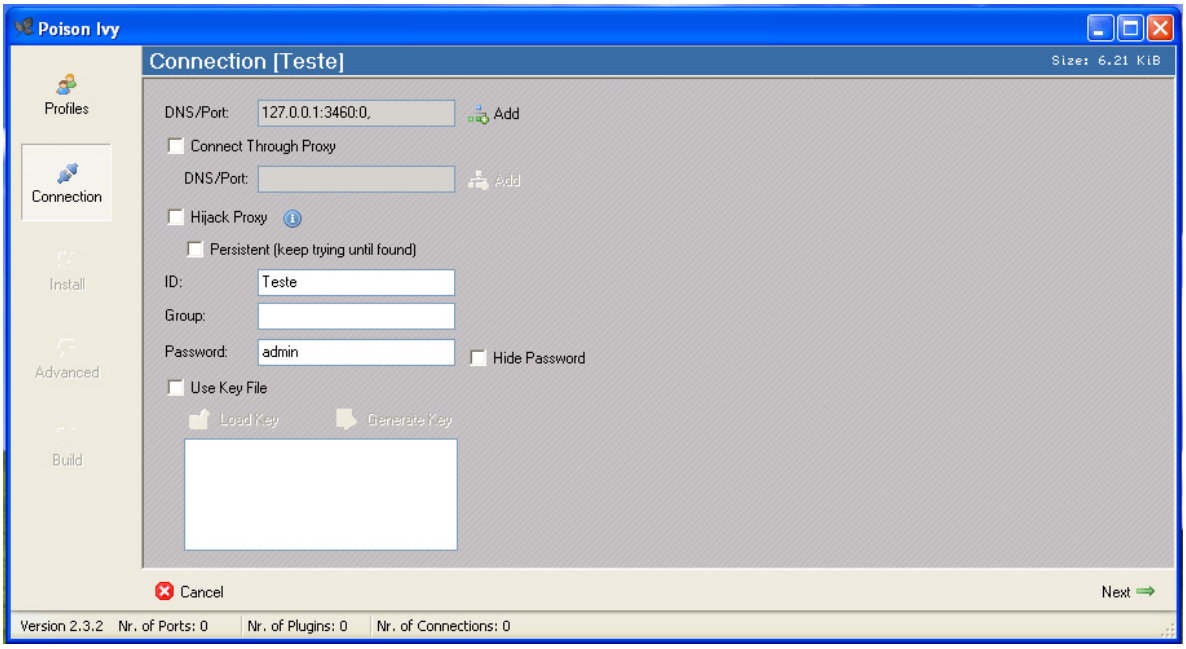



Figura A.1.6: Configuração da conexão do servidor

Como *default* o IP do servidor é o de *loopback*, 127.0.0.1, e a porta utilizada é a 3460, caso queira alterar essas configurações, é preciso clicar no botão "Add" ao lado do campo que exibe as configurações de IP e porta. A figura A.1.7 ilustra a nova janela para configuração desses parâmetros.

Lembrando que no caso do RAT, a conexão é iniciada pelo servidor com a execução do arquivo gerado nesta instalação, portanto há a necessidade de configurar o IP do cliente (atacante) e a porta utilizada para a comunicação.

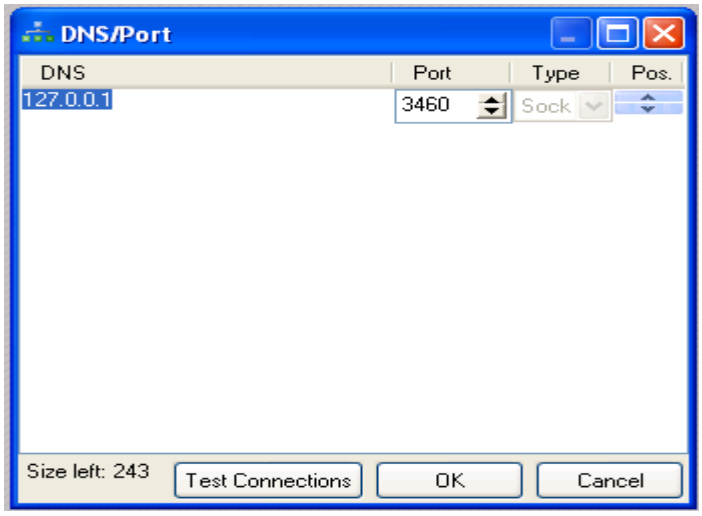



Figura A.1.7: Configuração de endereço IP e porta do servidor

Depois de configurar o modo de conexão que o servidor utilizará, o próximo passo será instalar o servidor na máquina a ser infectada, para isso a ferramenta apresenta algumas opções de configurações, figura A.1.8, tais como, iniciar junto com o sistema operacional, adicionar sua execução no registro do Windows e criar cópia do arquivo executável para pasta System32 ou Windows, podendo deletar o arquivo original ou não (opção melt).

É necessário atribuir um nome para o arquivo do executável no campo “Copy File” com umas das extensões: exe, scr, bat ou com, caso seja habilitado para copiar para outra pasta, pois sem isso não será possível concluir a instalação.

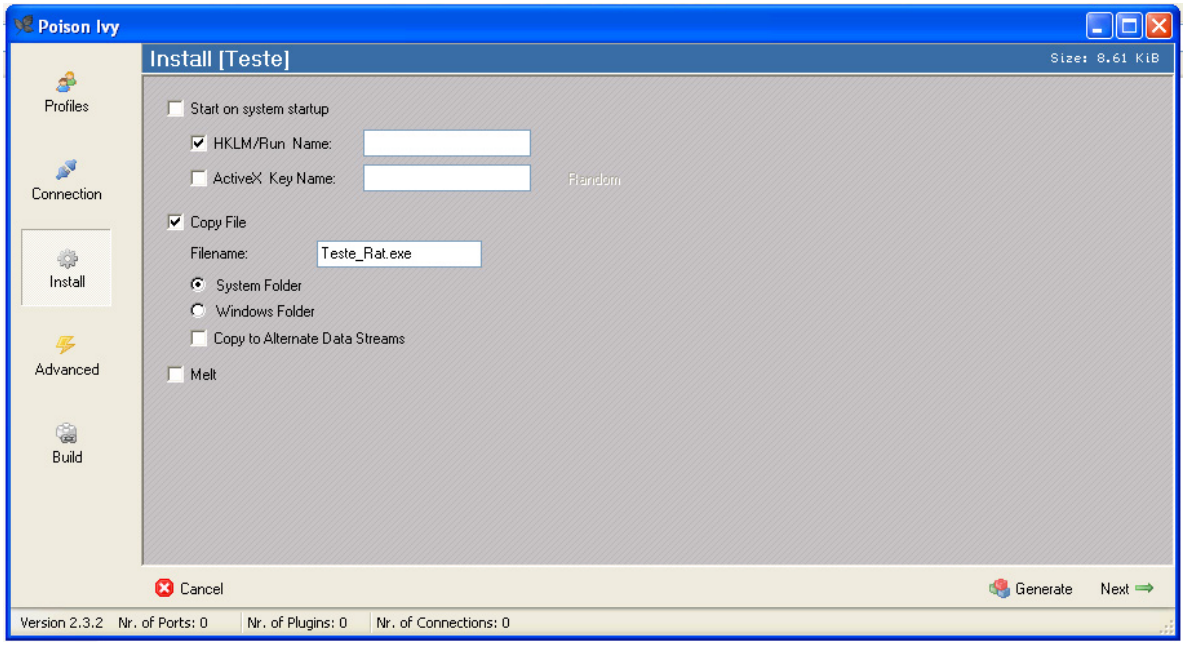



Figura A.1.8: Opções de instalação do servidor

Continuando, na próxima tela, figura A.1.9, há algumas opções avançadas de configuração, como por exemplo, associar sua execução a processos legítimos do computador, e para ofuscar seu código, pode ter como saída diversos formatos de código fonte: PE, binário, Delphi, Python, etc.

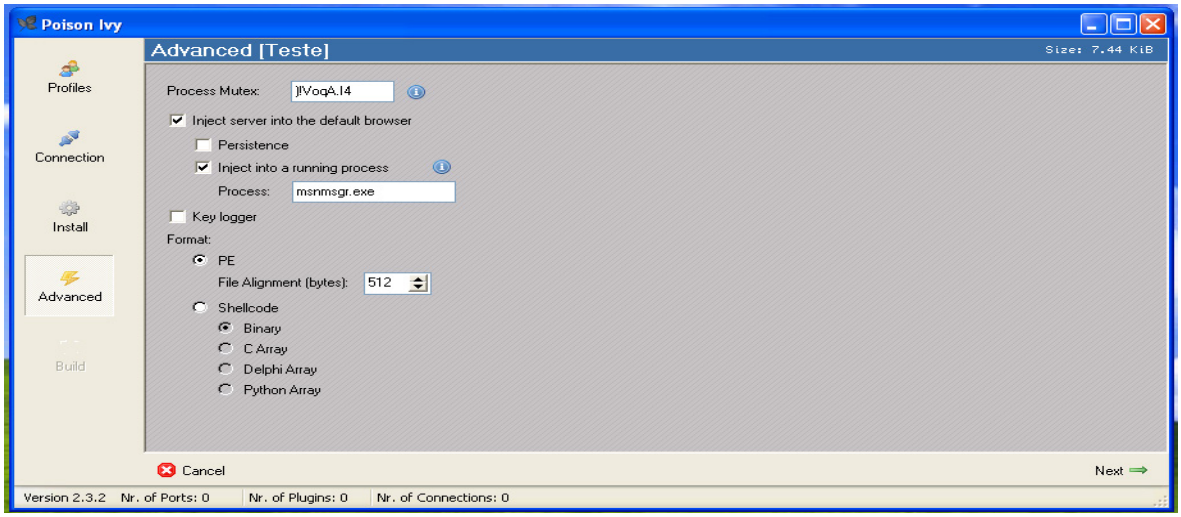



Figura A.1.9: Opções avançadas de instalação do servidor

Depois, na tela seguinte, figura A.1.10, há a possibilidade de se inserir ícone para o executável do RAT e executar outros programas junto com o servidor. Para concluir a instalação, é só clicar na opção “Generate”.

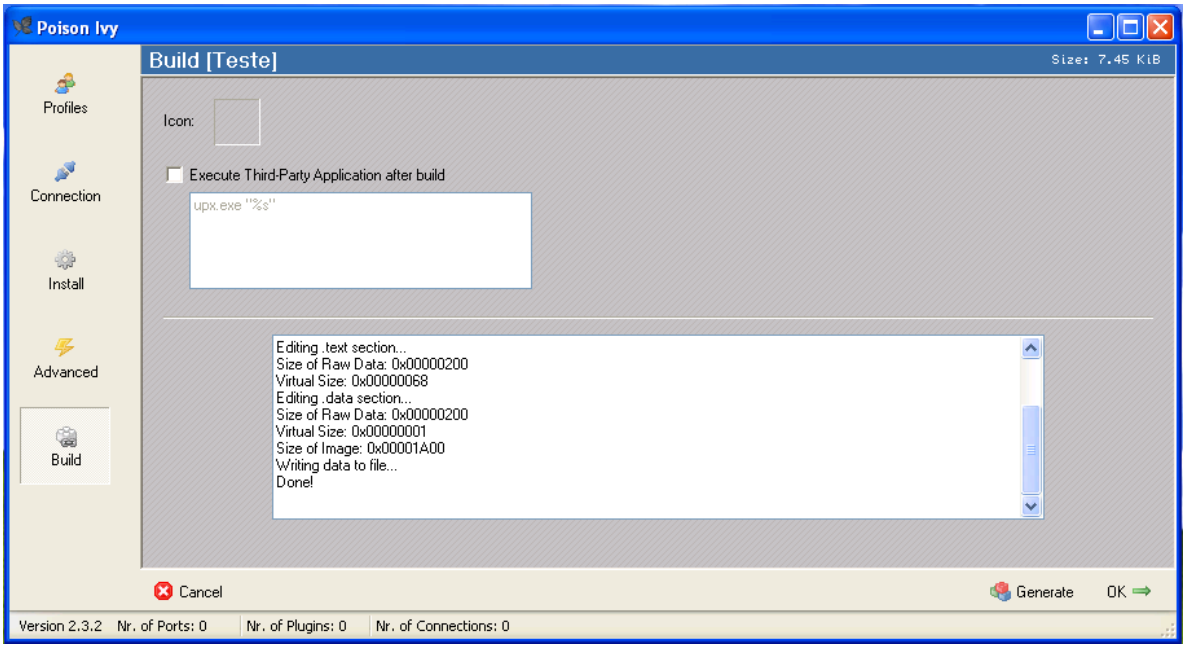
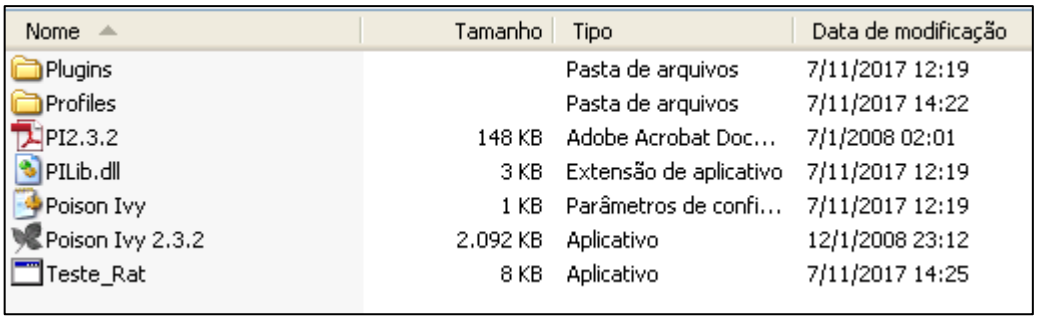



Figura A.1.10: Finalização da criação do servidor

Por fim, após ter criado o servidor, pode-se ver na figura A.1.11, os arquivos gerados, que incluí a configuração do perfil do servidor e o arquivo Teste_Rat.exe, o qual é o executável do servidor gerado.

Normalmente essa etapa de configuração é elaborada pelo atacante em sua máquina, a partir disso ele pode disseminar o servidor pode diversos vetores de ataque, com intuito de executar o servidor para inciar a comunicação.



| Nome | Tamanho | Tipo | Data de modificação |
|---|---|---|---|
| Plugins | | Pasta de arquivos | 7/11/2017 12:19 |
| Profiles | | Pasta de arquivos | 7/11/2017 14:22 |
| PI2.3.2 | 148 KB | Adobe Acrobat Doc... | 7/1/2008 02:01 |
| PILib.dll | 3 KB | Extensão de aplicativo | 7/11/2017 12:19 |
| Poison Ivy | 1 KB | Parâmetros de confi... | 7/11/2017 12:19 |
| Poison Ivy 2.3.2 | 2.092 KB | Aplicativo | 12/1/2008 23:12 |
| Teste_Rat | 8 KB | Aplicativo | 7/11/2017 14:25 |

Figura A.1.11: Arquivos gerados após a criação do servidor

## A.2 Instalação do cliente

Para ativar o cliente do Poison Ivy, utiliza-se o mesmo aplicativo da figura A.1.1, porém em vez de escolher a opção “New Server”, deve-se selecionar a opção “New Client”, conforme figura A.2.1.

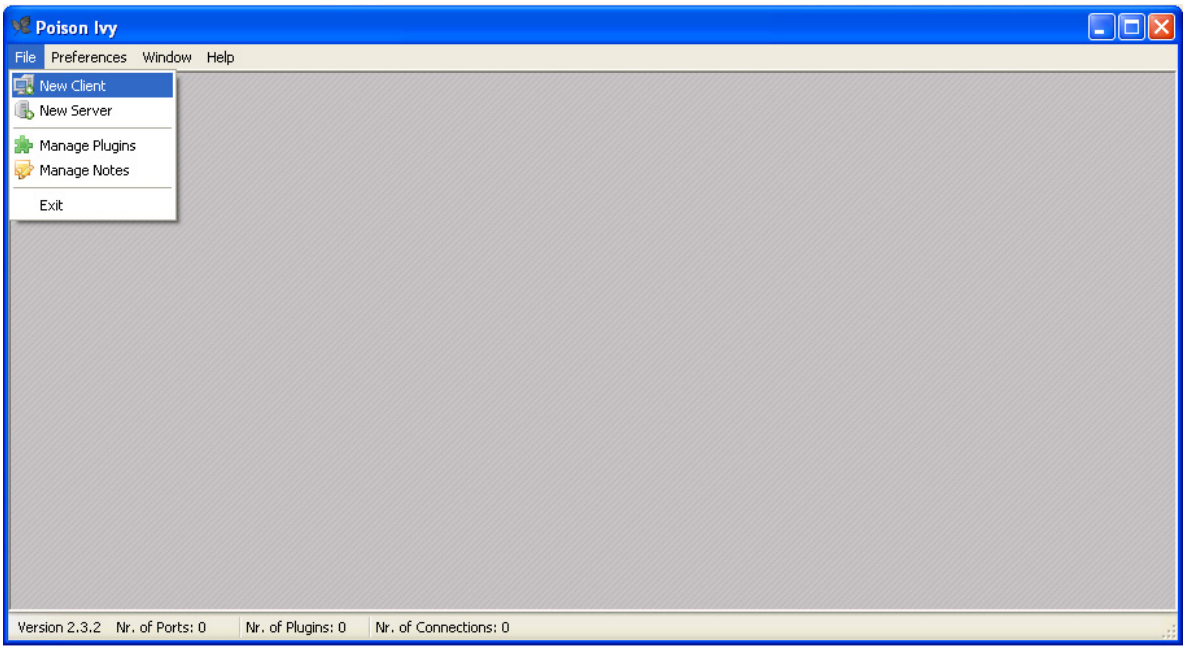



Figura A.2.1: Criar cliente

Após a seleção da opção cliente, será exibida a tela de configuração do cliente, figura A.2.2, devendo ser replicada a configuração criada no servidor. Percebe-se que apenas os campos porta de escuta e a autenticação são editáveis.

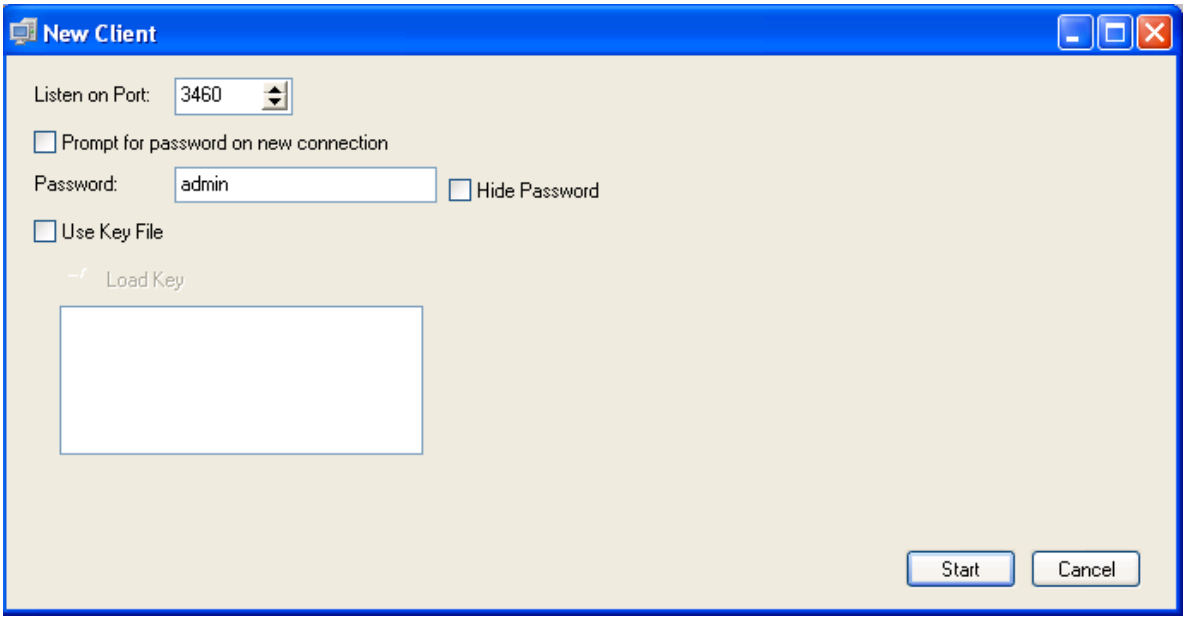



Figura A.2.2: Configuração do cliente

Uma vez iniciado o cliente após selecionar a opção “Start” da figura A.2.2, a máquina do atacante abrirá a porta definida na configuração para escutar conexões. A figura A.2.3 exibe as portas abertas na máquina Windows do atacante, neste exemplo foi definida a porta 3460.

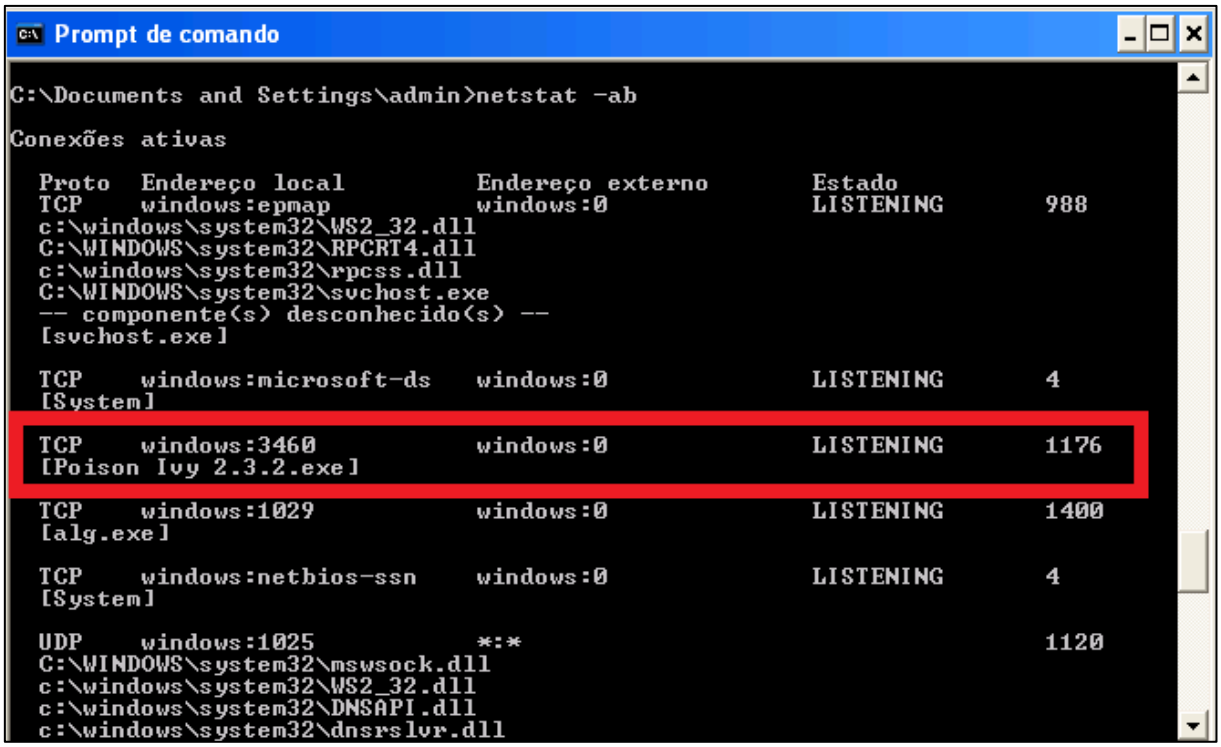



Figura A.2.3: Portas abertas no atacante

Agora resta executar o arquivo servidor para iniciar a comunicação com o atacante. A figura A.2.4 exibe a tela de conexões ativas do Poison Ivy, observa-se que na barra de status o número de portas abertas neste momento é igual a 1.

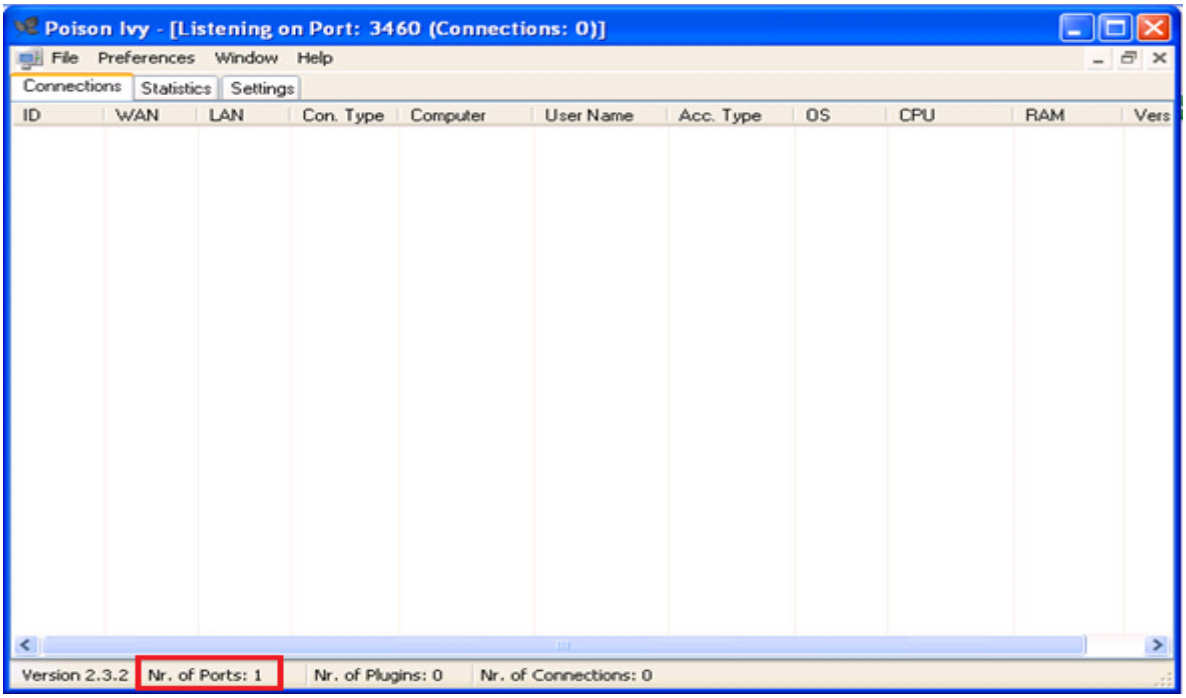



Figura A.2.4: Conexões ativas do Poison Ivy

De acordo com a figura A.2.5, após a execução do arquivo na máquina infectada, tem-se o alerta de conexão na tela de conexões ativas, bem como a opção de selecionar qual máquina conectada acessar.

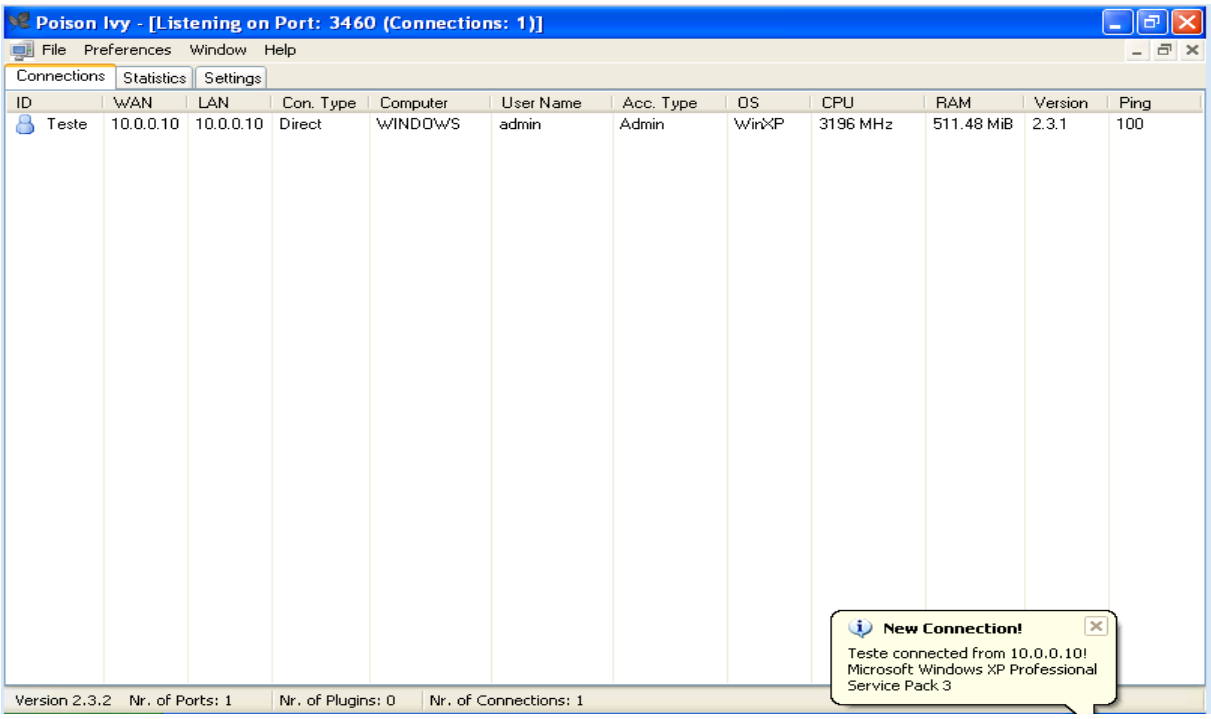



Figura A.2.5: Conexão estabelecida

## A.3 Exemplo de ataque

Com a conexão estabelecida, pode-utilizar os recursos disponíveis nesse RAT, a figura A.3.1 mostra essas ferramentas. Foi utilizada a opção de gravar a tela da máquina infectada para ilustar o uso.

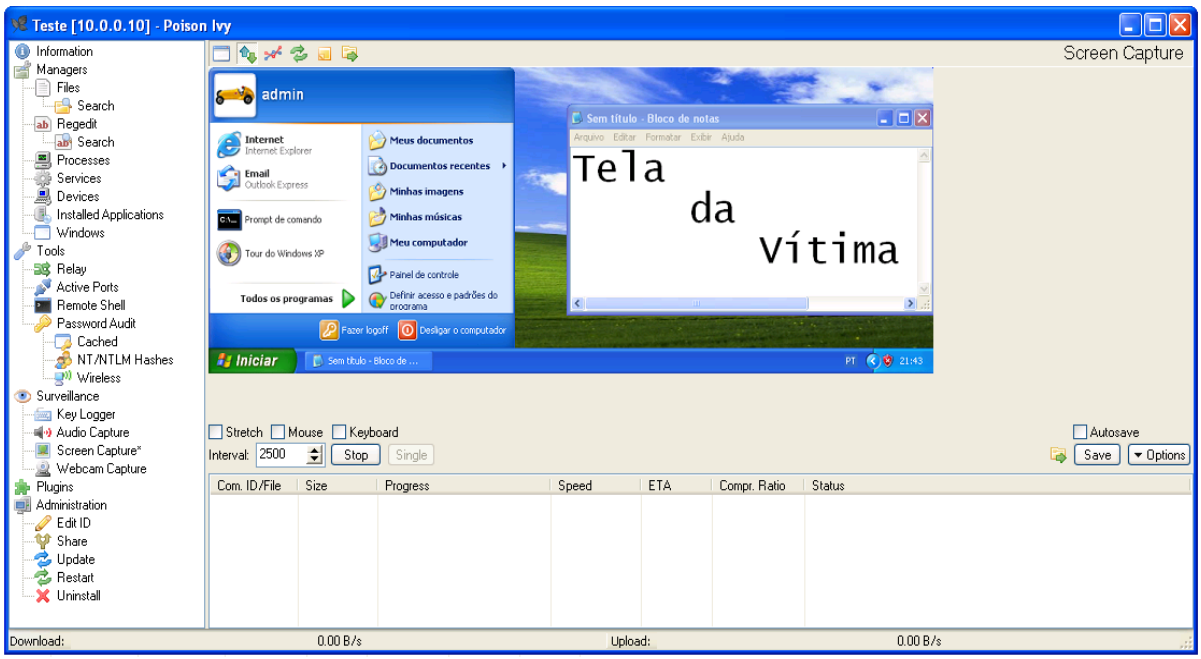



Figura A.3.1: Uso das ferramentas do Poison Ivy 2.3.2

# APÊNDICE B – Shell script para extração das características

```
#!/bin/bash
# Extração da captura
#Armazena a data e hora do início e término da captura, e o total de pacotes
capinfos -T -a -e -c $1 | tr -s ',' '.' | tr -s "\t" ',' > Informacao_Captura.csv
#leitura da linha com os valores das informações
Linha_info=$(tail -n +2 Informacao_Captura.csv)
Quantidade_Pacotes=$(echo $Linha_info |awk -F',' '{print $2}')
hora_inicio=$(echo $Linha_info |awk -F',' '{print $3}')
hora_termino=$(echo $Linha_info |awk -F',' '{print $4}')
#Executa a estatística conversation do tshark - estatística das conexões TCP estabelecidas
tshark -q -t r -z conv,tcp -r $1 > temp
#Cria cabeçalho para o CSV com estatísticas gerada
echo
"IP_Cliente,Porta_Cliente,IP_Servidor,Porta_Servidor,Segmentos_Enviados,Bytes_Enviados,
Segment
os_Recebidos,Bytes_Recebidos,Total_Segmentos,Total_Bytes,Tempo_relativo_inicio_conex
ao,Duracao
" > Estatistica_Inicial.csv
# Formata os dados estatísticos, pois o resultado do comando é um texto com vários
caracteres desnecessários
sed -i '/=/d' temp
tail -n +5 temp | tr -s " " | awk '{gsub("<->","",$0)}1' | awk '{gsub(":"," ",$1)}1' | awk
'{gsub(":"," ",$3)}1' | awk '{gsub(","",".",$11)}1' | awk '{gsub(","",".",$12)}1' | awk '{gsub("
","","")}1' >> Estatistica_Inicial.csv
#Exclui o arquivo temp criado, pois não será mais utilizado
rm -r temp
#Extrai os valores e informações escolhidas dos campos de todos os pacotes
#Exporta para formato CSV
tshark -T fields -e frame.time_relative -e tcp.stream -e tcp.flags.push -e ip.ttl -E header=n –E
separator=, -r $1 > Campos_Pacotes.csv
#Exporta para formato JSON
tshark -T json -e frame.time_relative -e tcp.stream -e tcp.flags.push -e ip.ttl -r $1 >
Campos_Pacotes.json
#Exporta formato JSON para elasticsearch
```

```
tshark -T ek -e frame.time_relative -e tcp.stream -e tcp.flags.push -e ip.ttl -r $1 >
Campos_Pacotes_elasticsearch.json
#Adiciona cabeçalho ao arquivo CSV com os dados dos pacotes
sed -i '1s/^/Tempo_Relativo_Captura,Numero_TCP_Stream,Flag_Push_Setada,TTL\n/'
Campos_Pacotes.csv
exit 0
```

# APÊNDICE C – Classe Scala - tratamento dos arquivos CSV

```
/**
* Autor: Felipe
*
* Referências:
* https://spark.apache.org/docs/2.1.0/sql-programming-guide.html
* https://spark.apache.org/docs/2.1.0/api/scala/
* https://spark.apache.org/docs/1.5.1/api/scala/index.html#org.apache.spark.sql.DataFrame
* Data Science Academy - http://www.datascienceacademy.com.br
*
* Manipulação do CSV com Spark SQL
*/
class TratamentoCSV {

/*
Importa as bibliotecas, define o nível de log e cria variável do Spark e diretório dos arquivos
*/
// Importando pacotes
import org.apache.spark.sql.functions._
import org.apache.spark.sql.Row
import org.apache.spark.sql.types._
import org.apache.log4j.{Level, Logger}

// Definindo nível de log
Logger.getLogger("org").setLevel(Level.ERROR)
Logger.getLogger("akka").setLevel(Level.ERROR)

// Definindo variáveis a partir da classe ConfiguracaoSpark
val spSession = ConfiguracaoSpark.sparkSess
val spContext = ConfiguracaoSpark.sparkCon
val datadir = ConfiguracaoSpark.datadir

/*
Importando o CSV com as estatisticas das conexoes
*/

//Cria o esquema do dataframe do CSV Estatísticas
val schema =
StructType(

/*
identifica a conexão
*/

StructField("Cliente", StringType, nullable = false) ::
```

```
StructField("C_Porta", IntegerType, nullable = false) ::
StructField("Servidor", StringType, nullable = false) ::
StructField("S_Porta", IntegerType, nullable = false) ::
/*
Caracteristicas das conexoes
Legendas:
SES - Segmentos_Enviados_Servidor
BES - Bytes_Enviados_Servidor
SEC - Segmentos_Enviados_Cliente
BEC - Bytes_Enviados_Cliente
TS - Total_Segmentos
TB - Total_Bytes
*/
StructField("SES", DoubleType, nullable = false) ::
StructField("BES", DoubleType, nullable = false) ::
StructField("SEC", DoubleType, nullable = false) ::
StructField("BEC", DoubleType, nullable = false) ::
StructField("TS", DoubleType, nullable = false) ::
StructField("TB", DoubleType, nullable = false) ::

// identifica a conexão
StructField("Tempo_Relativo", DoubleType, nullable = false) ::

//caracteristica das conexões
StructField("Duracao", DoubleType, nullable = false) ::
StructField("BES/SES", DoubleType, nullable = false) ::
StructField("BEC/SEC", DoubleType, nullable = false) ::
StructField("BES/BEC", DoubleType, nullable = false) ::
StructField("SES/SEC", DoubleType, nullable = false) ::
StructField("TS/Duracao", DoubleType, nullable = false) ::
StructField("TB/Duracao", DoubleType, nullable = false) ::
Nil)

// Cria dataframe a partir do arquivo csv estatísticas das conexões
val DF_CSV_Estatisticas = spSession.read.option("header", "true").schema(schema).csv(
datadir + "Estatistica_Inicial.csv")
/*
Adiciona colunas derivadas das originais do CSV
Legenda:
BES/SES - total de bytes enviados pelo servidor / quantidade de segmentos enviados pelo
servidor - media de bytes por segmentos enviados pelo servidor
BEC/SEC - total de bytes enviados pelo cliente /quantidade de segmentos enviados pelo
cliente - media de bytes por segmentos enviados pelo cliente
BES/BEC - taxa de bytes enviados servidor / taxa de bytes enviados cliente
SES/SEC - taxa de segmentos enviados servidor / taxa de segmentos enviados cliente
TS/Duracao - total de segmentos da conexao / duracao da conexao - taxa de segmentos por
segundos da conexao
TB/Duracao - total de bytes da conexao / duracao da conexao - taxa de bytes por segundos da
conexao
*/
```

```
val DF_CSV_Novo = DF_CSV_Estatisticas.withColumn("BES/SES",
(DF_CSV_Estatisticas("BES").
divide(DF_CSV_Estatisticas("SES")))).withColumn("BEC/SEC",
(DF_CSV_Estatisticas("BEC").
divide(DF_CSV_Estatisticas("SEC")))).withColumn("BES/BEC",
(DF_CSV_Estatisticas("BES").
divide(DF_CSV_Estatisticas("BEC")))).withColumn("SES/SEC",
(DF_CSV_Estatisticas("SES").
divide(DF_CSV_Estatisticas("SEC")))).withColumn("TS/Duracao",
(DF_CSV_Estatisticas("TS").
divide(DF_CSV_Estatisticas("Duracao")))).withColumn("TB/Duracao",
(DF_CSV_Estatisticas("TB"
).divide(DF_CSV_Estatisticas("Duracao"))))


//remove da memoria o dataframe antigo para ser excluído, pois não será mais utilizado
DF_CSV_Estatisticas.unpersist()

/*
Normaliza os dados para uma característica não influenciar no resultado mais do que as
outras, conforme equacão 1
*/

//Consulta os valores máximos e mánimos de cada coluna com características
val Valores_maximos = DF_CSV_Novo.agg(max("SES"), max("BES"), max("SEC"),
max("BEC"), max(
"TS"), max("TB"), max("Duracao"), max("BES/SES"), max("BEC/SEC"), max("BES/BEC"),
max(
"SES/SEC"), max("TS/Duracao"), max("TB/Duracao")).collectAsList().get(0)
val Valores_minimos = DF_CSV_Novo.agg(min("SES"), min("BES"), min("SEC"),
min("BEC"), min(
"TS"), min("TB"), min("Duracao"), min("BES/SES"), min("BEC/SEC"), min("BES/BEC"),
min(
"SES/SEC"), min("TS/Duracao"), min("TB/Duracao")).collectAsList().get(0)

//funcao para normalizar os valores entre 0 e 1, de acordo com a equacão 1
def normalizar(a: String, b: Int): org.apache.spark.sql.Column = {
val CalulaNormalizacao = DF_CSV_Novo(a).minus(Valores_minimos.getDouble(b))./(
Valores_maximos.getDouble(b) - Valores_minimos.getDouble(b))

return CalulaNormalizacao
}

//Aplica a normalizacão em todas as colunas
val normalizando = DF_CSV_Novo.withColumn("SES", (normalizar("SES",
0))).withColumn("BES", (
normalizar("BES", 1))).withColumn("SEC", (normalizar("SEC", 2))).withColumn("BEC", (
normalizar("BEC", 3))).withColumn("TS", (normalizar("TS", 4))).withColumn("TB",
(normalizar
("TB", 5))).withColumn("Duracao", (normalizar("Duracao", 6))).withColumn("BES/SES", (
```

```
normalizar("BES/SES", 7))).withColumn("BEC/SEC", (normalizar("BEC/SEC",
8))).withColumn(
"BES/BEC", (normalizar("BES/BEC", 9))).withColumn("SES/SEC",
(normalizar("SES/SEC", 10))).
withColumn("TS/Duracao", (normalizar("TS/Duracao", 11))).withColumn("TB/Duracao", (
normalizar("TB/Duracao", 12)))

//caso a divisao gere valores null ou NaN, preenche com o valor 0
val normalizado = normalizando.na.fill(0)

//seleciona as colunas para ordenar diferente da estatística gerada pelo tshark, a fim de
facilitar a visualização
val DF_CSV_Estatisticas_normalizado =
normalizado.select(normalizado("Tempo_Relativo"),
normalizado("Servidor"), normalizado("S_Porta"), normalizado("Cliente"), normalizado(
"C_Porta"), normalizado("SES"), normalizado("BES"), normalizado("SEC"),
normalizado("BEC"),
normalizado("TS"), normalizado("TB"), normalizado("Duracao"), normalizado("BES/SES"),
normalizado("BEC/SEC"), normalizado("BES/BEC"), normalizado("SES/SEC"),
normalizado(
"TS/Duracao"), normalizado("TB/Duracao"))

//remove da memoria o dataframe antigo para ser excluído, pois não será mais utilizado
DF_CSV_Novo.unpersist()
normalizado.unpersist()



/*
Calcula caracteristicas do CSV campos pacotes
*/


/*
Importando o CSV com os campos dos pacotes escolhidos
*/

//Cria o esquema do dataframe do CSV Campos_pacotes
val schema_campos =
StructType(
//identifica o pacote
StructField("Tempo_Relativo", DoubleType, nullable = false) ::
StructField("TCP_Stream", IntegerType, nullable = false) ::
//caracteristica
StructField("Flag_PUSH", IntegerType, nullable = false) ::
StructField("TTL", IntegerType, nullable = false) ::
Nil)

// Cria dataframe a partir do arquivo csv campos das conexoes
```

```
val DF_CSV_Campos = spSession.read.option("header",
"true").schema(schema_campos).csv(
datadir + "Campos_Pacotes.csv")

//Calcula o valor máximo de linhas do dataframe de estatisticas, para determinar quantas
conexões foram estabelecidas
val Total_conexoes = DF_CSV_Estatisticas_normalizado.count().toInt
val schema_campos_calculados =
StructType(
//identifica o pacote
StructField("Tempo_Relativo2", DoubleType, nullable = false) ::
//caracteristicas
StructField("%_PUSH", DoubleType, nullable = false) ::
StructField("Media_TTL", DoubleType, nullable = false) ::
Nil)
//Método para agrupar os pacotes por conexão e calcular as características
def CalculaConexao(x: Int, y: Any): org.apache.spark.sql.DataFrame = {

//armazena o tempo de inicio da conexao
var Tempo_Relativo = y

//agrupa por TCP Stream
var conexao = DF_CSV_Campos.filter(DF_CSV_Campos("TCP_Stream") === x)

//calcula media dos valores do TTL
var Stats_TTL = conexao.agg(avg("TTL"), min("TTL"), max("TTL")).collectAsList().get(0)
var Media_TTL = Stats_TTL.getDouble(0)

//normaliza os valores
var Min_TTL = Stats_TTL.getInt(1)
var Max_TTL = Stats_TTL.getInt(2)
var TTL = 0.0
//Evita com denominador igual a zero
if (Max_TTL - Min_TTL > 0) {
TTL = (Media_TTL - Min_TTL) / (Max_TTL - Min_TTL)
}

//Calcula a porcentagem de Push na conexao
var Stats_Push = conexao.agg(sum("Flag_PUSH"), count("Flag_PUSH")).collectAsList().get(
0)
var Soma_Push = Stats_Push.getLong(0)
var Quantidade_Push = Stats_Push.getLong(1)
var Porcentagem_PUSH = Soma_Push./(Quantidade_Push.toDouble)

//lista com os valores calculados
var Lista_valores = Array(Row(Tempo_Relativo, Porcentagem_PUSH, TTL))
//converte para dataframe
var rdd = spContext.parallelize(Lista_valores)
var conexao_calculada = spSession.createDataFrame(rdd, schema_campos_calculados)
return conexao_calculada
```

```
}
var DF_CSV_Campos_Calculados =
spSession.createDataFrame(spContext.emptyRDD[Row],
schema_campos_calculados)

//loop para calcular todas as conexoes
for (i <- 0 to (Total_conexoes-1)) {

//ler linha por linha o tempo relativo de cada conexao
var Tempo_Relativo = DF_CSV_Estatisticas_normalizado.collect().apply(i).get(0)

//identifica o TCP Stream de cada conexão, conforme o tempo relativo de inicio da conexao
val TCP_Stream = DF_CSV_Campos.filter(DF_CSV_Campos("Tempo_relativo").equalTo(
Tempo_Relativo))
.select("TCP_Stream").collectAsList().get(0).getInt(0)

//adiciona nova linha ao dataframe
var temp =
DF_CSV_Campos_Calculados.union(CalculaConexao(TCP_Stream,Tempo_Relativo))
DF_CSV_Campos_Calculados = temp
}

//Junta os dois dataframes com caracteristicas
val Join_Dataframe =
DF_CSV_Estatisticas_normalizado.join(DF_CSV_Campos_Calculados,
DF_CSV_Estatisticas_normalizado("Tempo_relativo") === DF_CSV_Campos_Calculados(
"Tempo_relativo2"))
//remove coluna repetida
val Resultado_Calculo_Caracteristicas = Join_Dataframe.drop("Tempo_Relativo2")
Resultado_Calculo_Caracteristicas.show()

//remove da memoria dataframe antigo
Join_Dataframe.unpersist()



//calcula o número de servicos, ao verificar a quantidade de portas distintas do servidor
val servicos = DF_CSV_Estatisticas_normalizado.select("S_Porta").distinct().count()
}
```

# APÊNDICE D – Object Scala - configuração do Spark

```
/**
* Autor: Felipe
*
*
* Referências:
* https://spark.apache.org/docs/latest/submitting-applications.html
* https://spark.apache.org/docs/latest/configuration.html
* https://spark.apache.org/docs/2.1.0/spark-standalone.html
* Data Science Academy - http://www.datascienceacademy.com.br
*
* Definição de utilitários usados pelos algoritmos de Machine Learning com Scala e Spark
*/
object ConfiguracaoSpark {
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.sql.SparkSession

// Diretório onde estão os arquivos CSV
val datadir = "./ArquivosCSV/"

// Nome da instância Spark
val appName = "SparkRAT"

// Configura o Spark para executar standalone com paralelismo dos processadores disponiveis
val sparkMasterURL = "local[*]"

// Diretório temporário requerido pelo Spark SQL
val tempDir = "/tmp/spark-warehouse"

//Variaveis para criar sessoes e contextos Spark
var sparkSess:SparkSession = null
var sparkCon:SparkContext = null

// Inicialização - Executa quando o objeto é criado
{
// É necessário configurar hadoop.home.dir para evitar erros na inicialização do Spark
System.setProperty("hadoop.home.dir","/tmp/hadoop")

// Objeto de configuração do Spark
val conf = new SparkConf()
.setAppName(appName)
.setMaster(sparkMasterURL)
.set("spark.executor.memory","20g")
.set("spark.driver.memory","20g")
```

```
// Get ou create Spark context. Cria uma nova instância se não existir uma disponível.
sparkCon = SparkContext.getOrCreate(conf)

// Cria sessão Spark SQL
sparkSess = SparkSession
.builder()
.appName(appName)
.master(sparkMasterURL)
.config("spark.sql.warehouse.dir", tempDir)
.getOrCreate()
}
}
```

# APÊNDICE E – Object Scala - modelagem dos grupos com o Kmeans

```
import org.apache.spark.ml.{Pipeline, PipelineModel}
import org.apache.spark.sql.Row
import org.apache.spark.ml.clustering.{KMeans, KMeansModel}
import org.apache.spark.ml.feature.{OneHotEncoder, StringIndexer, VectorAssembler,
VectorIndexer}
import org.apache.log4j.{Level, Logger}
import org.apache.spark.sql.functions._
/**
* Autor: Felipe
*
* Referências:
*
https://github.com/sryza/aas/blob/master/ch05-
kmeans/src/main/scala/com/cloudera/datascience
/kmeans/RunKMeans.scala
* https://spark.apache.org/docs/2.1.1/mllib-clustering.html
* Data Science Academy - http://www.datascienceacademy.com.br
*
* Algoritmo K-means com os dados de saida do estágio de tratamento dos dados
*/
object Kmeans extends App {

// Nível de log
Logger.getLogger("org").setLevel(Level.ERROR)
Logger.getLogger("akka").setLevel(Level.ERROR)

// Variáveis
val spSession = ConfiguracaoSpark.sparkSess
val spContext = ConfiguracaoSpark.sparkCon
val datadir = ConfiguracaoSpark.datadir

//Cria instancia da classe TratamentoCSV
val Tratar_dados = new TratamentoCSV

//Cria dataframe com as estatísticas normalizadas
val Dados_conexoes = Tratar_dados.DF_CSV_Estatisticas_normalizado

//cria variavel responsável para atribuir o número de clusters do Kmeans (K)
val Valor_K = Tratar_dados.servicos.toInt

//Define as caracteristicas escolhidas para criar os grupos
val cluster0 = Seq[String] ("BEC/SEC", "TB")
val cluster1 = Seq[String] ("BEC","TB/Duracao")
val cluster2 = Seq[String] ("TS","SES","BES/BEC")
```

```
val cluster3 = Seq[String] ("BES/SES","BES","SES/SEC")
val cluster4 = Seq[String] ("SEC","TS/Duracao","Duracao")

//cria Row para adicionar outro grupo no loop for
val cluster = Row (cluster0,cluster1,cluster2,cluster3,cluster4)

//metodo para executar clustering com o kmeans
def Clustering (coluna: Seq[String], cluster: String, dataframe: org.apache.spark.sql.
DataFrame) : org.apache.spark.sql.DataFrame = {

//Seleciona as colunas com as características de aprendizagem
var caracteristicas = dataframe.select(coluna.head, coluna.tail: _*)

//Cria vetor com todas as caracteristicas, isso é necessário para o kmeans
var assembler = new VectorAssembler().
setInputCols(caracteristicas.columns).
setOutputCol("VetorCaracteristicas")


//Cria o Kmeans, atribuindo o valor de K e configurando os seus parâmetros
var kmeans = new KMeans().
setK(Valor_K).
setSeed(1L).
setMaxIter(1).
setPredictionCol(cluster).
setFeaturesCol("VetorCaracteristicas")

//Divide o dataframe de entrada em dois, um para treinar o algoritmo (90% dos dados)
e outro de teste para avaliar o modelo (10% dos dados)
var Array(trainingData, testData) = dataframe.randomSplit(Array(0.9, 0.1), 5043)

//Usado para combinar dataframes com algoritmo de machine learning
var pipeline = new Pipeline().setStages(Array(assembler, kmeans))

//Cria o modelo com o dataframe de treinamento
var pipelineModel = pipeline.fit(trainingData)
var kmeansModel = pipelineModel.stages.last.asInstanceOf[KMeansModel]

//Salva os modelos dos grupos de características
pipelineModel.write.save(datadir+cluster)
var Dados_Teste = pipelineModel.transform(testData)

//testa a eficiência do modelo criado com WSSSE
var WSSSE = kmeansModel.computeCost(Dados_Teste)
println (cluster)
println(s"Soma erro quadratico = $WSSSE")

//Retira a coluna criada com o vetor de características, pois não será mais utilizada
var Resultado = Dados_Teste.drop("VetorCaracteristicas")
```

```
//retorna todo o dataframe com a nova coluna do grupo de características
return Resultado
}
//Chama o método do Kmeans para criar a coluna com o primeiro grupo de características
var temp = Clustering(cluster.getSeq(0), "cluster"+0, Dados_conexoes)

//seleciona apenas a coluna com o tempo relativo e a coluna com o grupo de
características, renomeia a coluna Tempo_relativo para facilitar a sua retirada após o
join
var temp_coluna = temp.select("Tempo_relativo", "cluster" + 0).withColumnRenamed(
"Tempo_relativo","Tempo_relativo2")

//Adiciona a coluna com o grupo de características
var Resultado_clustering = Dados_conexoes.join(temp_coluna, Dados_conexoes(
"Tempo_relativo")=== temp_coluna ("Tempo_relativo2")).drop("Tempo_relativo2")

//loop para criar todos os modelos do Kmeans
for (x <- 1 to 4) {
temp = Clustering(cluster.getSeq(x), "cluster" + x, Dados_conexoes)
temp_coluna = temp.select("Tempo_relativo", "cluster" + x).withColumnRenamed(
"Tempo_relativo","Tempo_relativo2")
Resultado_clustering = Resultado_clustering.join(temp_coluna, Resultado_clustering(
"Tempo_relativo")=== temp_coluna ("Tempo_relativo2")).drop("Tempo_relativo2")
}

//Exibe o resultado final para verificar se todas as colunas foram criadas no dataframe
de teste
Resultado_clustering.show()
}
```

# APÊNDICE F – Object Scala – Aprendizagem Random Forest

```
package DeteccaoRAT
import org.apache.spark.ml.{Pipeline, PipelineModel}
import org.apache.spark.ml.clustering.{KMeans, KMeansModel}
import org.apache.spark.ml.feature._
import org.apache.log4j.{Level, Logger}
import org.apache.spark.ml.classification.RandomForestClassifier
import org.apache.spark.sql.functions._
import org.apache.spark.mllib.evaluation.MulticlassMetrics
/**
* Autor: Felipe
*
* Referencias:
* https://spark.apache.org/docs/2.2.0/ml-classification-regression.html
* https://spark.apache.org/docs/2.2.0/mllib-ensembles.html
*
http://blog.learningtree.com/how-to-build-a-random-forest-classifier-using-data-frames-in-spark/
* https://spark.apache.org/docs/2.2.0/mllib-evaluation-metrics.html
* Data Science Academy - http://www.datascienceacademy.com.br
*
* Treinamento do algoritmo Random Forest
*/
object Random_Forest_Aprendizagem extends App {
// Nível de log
Logger.getLogger("org").setLevel(Level.ERROR)
Logger.getLogger("akka").setLevel(Level.ERROR)
// Variáveis
val spSession = ConfiguracaoSpark.sparkSess
val spContext = ConfiguracaoSpark.sparkCon
val datadir = ConfiguracaoSpark.datadir
//Cria instância das classes de TratamentoCSV
val Tratar_dados = new TratamentoCSV
val Tratar_dados_RAT = new TratamentoCSV_RAT
//Cria dataframe com as estatísticas normalizadas
val Dados_conexoes = Tratar_dados.DF_CSV_Estatisticas_normalizado
val Dados_conexoes_RAT = Tratar_dados_RAT.DF_CSV_Estatisticas_normalizado
//cria variável responsável por atribuir o número de clusters do Kmeans (K)
val Valor_K = Tratar_dados.servicos.toInt
val Valor_K_RAT = Tratar_dados_RAT.servicos.toInt
//método para executar clustering com o k-means - Agora só executa os modelos nos dados
def Clustering(cluster: String, dataframe: org.apache.spark.sql.DataFrame): org.apache.
spark.sql.DataFrame = {
var testData = dataframe
//carrega o modelo criado para aplicar em novos dados
```

```
var pipelineModel = PipelineModel.read.load(datadir + cluster)
var kmeansModel = pipelineModel.stages.last.asInstanceOf[KMeansModel]
var Dados_Teste = pipelineModel.transform(testData)
//testa a eficiencia do modelo criado com WSSSE
var WSSSE = kmeansModel.computeCost(Dados_Teste)
println(cluster)
println(s"Soma erro quadratico = $WSSSE")
var Resultado = Dados_Teste.drop("VetorCaracteristicas")

return Resultado
}
//Cria o dataframe com os dados de conexoes normais, o loop varia de acordo com a
quantidade de clusters definidos, neste caso foram criados 12 grupos de características
var temp = Clustering("cluster" + 0, Dados_conexoes)
var temp_coluna = temp.select("Tempo_relativo", "cluster" + 0).withColumnRenamed(
"Tempo_relativo", "Tempo_relativo2")
var Resultado_clustering = Dados_conexoes.join(temp_coluna, Dados_conexoes(
"Tempo_relativo") === temp_coluna("Tempo_relativo2")).drop("Tempo_relativo2")
for (x <- 1 to 12) {
temp = Clustering("cluster" + x, Dados_conexoes)
temp_coluna = temp.select("Tempo_relativo", "cluster" + x).withColumnRenamed(
"Tempo_relativo", "Tempo_relativo2")
Resultado_clustering = Resultado_clustering.join(temp_coluna, Resultado_clustering(
"Tempo_relativo") === temp_coluna("Tempo_relativo2")).drop("Tempo_relativo2")
}
//Cria o dataframe com os dados de conexoes RAT
temp = Clustering("cluster" + 0, Dados_conexoes_RAT)
temp_coluna = temp.select("Tempo_relativo", "cluster" + 0).withColumnRenamed(
"Tempo_relativo", "Tempo_relativo2")
var Resultado_clustering_RAT = Dados_conexoes_RAT.join(temp_coluna,
Dados_conexoes_RAT(
"Tempo_relativo") === temp_coluna("Tempo_relativo2")).drop("Tempo_relativo2")
for (x <- 1 to 12) {
temp = Clustering("cluster" + x, Dados_conexoes_RAT)
temp_coluna = temp.select("Tempo_relativo", "cluster" + x).withColumnRenamed(
"Tempo_relativo", "Tempo_relativo2")
Resultado_clustering_RAT = Resultado_clustering_RAT.join(temp_coluna,
Resultado_clustering_RAT("Tempo_relativo") === temp_coluna("Tempo_relativo2")).drop(
"Tempo_relativo2")
}
/*
Inicia a parte de aprendizagem do Random Forest
*/
//Cria coluna label com os valores 0 e 1
//dataframe com conexões normais
Resultado_clustering = Resultado_clustering.withColumn("label", lit(0))
//dataframe com conexões de RAT
Resultado_clustering_RAT = Resultado_clustering_RAT.withColumn("label", lit(1))
//Junta os dois dataframes
val Resultado_Final = Resultado_clustering.union(Resultado_clustering_RAT)
Resultado_Final.show()
```

```
//Seleciona as características para aprendizagem do Random Forest
var caracteristicas = Resultado_Final.select("SEC","BEC","TB","BES/BEC", "Duracao",
"BEC/SEC","TS/Duracao","TB/Duracao","TS","BES","SES","SES/SEC","BES/SES","cluste
r0",
"cluster1","cluster2","cluster3","cluster4","cluster5","cluster6","cluster7","cluster8",
"cluster9","cluster10","cluster11","cluster12")
//Cria vetor com todas as características
var assembler = new VectorAssembler().
setInputCols(caracteristicas.columns).
setOutputCol("features")
val Resultado_Final_caracteristicas = assembler.transform(Resultado_Final)

// Dividindo os dados em treino e teste com seed 5043
val Array(trainingData, testData) = Resultado_Final_caracteristicas.randomSplit(Array(
0.99, 0.01), 5043)
val classifier = new RandomForestClassifier()
.setImpurity("gini")
//profundidade da árvore
.setMaxDepth(30)
//número de árvores
.setNumTrees(128)
.setFeatureSubsetStrategy("auto")
.setSeed(5043)
//Treina o algoritmo
val model = classifier.fit(trainingData)
//Salva o modelo para ser utilizado posteriormente
model.write.overwrite().save(datadir+"RandomForest")
//varável para utilizar as métricas de desempenho do modelo
var predictions = model.transform(testData)
//exibe o resultado da predição, o label e as colunas que identificam a conexão
predictions.select("Tempo_relativo", "Servidor", "S_Porta","Cliente", "C_Porta", "label",
"prediction").show()
//seleciona as colunas com os valores do label e da predição
predictions = predictions.select("prediction","label")
import spSession.implicits._
//transforma as colunas para RDD, pois a classe MulticlassMetrics exige como entrada
var predictionsAndLabels = predictions.as[(Double,Double)].rdd
//Exibe a quantidade de conexões por label e sua predição
println("\nQuantidade por label:")
predictions.groupBy("label","prediction").count().show()
//Avaliação do modelo com a confusion matrix e as métricas de desempenho
val metrics = new MulticlassMetrics(predictionsAndLabels)
//Confusion matrix
println("Confusion Matrix:")
println(metrics.confusionMatrix.asML)
//Accracy do modelo
println("Accuracy:")
println(metrics.accuracy)
println()
// Precision por label
val labels = metrics.labels
```

```
labels.foreach { l =>
println(s"Precision($l) = " + metrics.precision(l))
}
println()
// Recall por label
labels.foreach { l =>
println(s"Recall($l) = " + metrics.recall(l))
}
println()
// F-measure por label
labels.foreach { l =>
println(s"F1-measure($l) = " + metrics.fMeasure(l))
}

}
```

# APÊNDICE G – Object Scala – Classificação do comportamento com Random Forest

```
package DeteccaoRAT
import org.apache.log4j.{Level, Logger}
import org.apache.spark.ml.PipelineModel
import org.apache.spark.ml.classification.{RandomForestClassificationModel,
RandomForestClassifier}
import org.apache.spark.ml.clustering.KMeansModel
import org.apache.spark.ml.feature._
import org.apache.spark.mllib.evaluation.MulticlassMetrics
import org.apache.spark.sql.functions._
/**
* Autor: Felipe
*
* Referencias:
* https://spark.apache.org/docs/2.2.0/ml-classification-regression.html
* https://spark.apache.org/docs/2.2.0/mllib-ensembles.html
*
http://blog.learningtree.com/how-to-build-a-random-forest-classifier-using-data-frames-in-sp
ark/
* https://spark.apache.org/docs/2.2.0/mllib-evaluation-metrics.html
* Data Science Academy - http://www.datascienceacademy.com.br
*
* Treinamento do algoritmo Random Forest
*/
object Random_Forest_Deteccao extends App {
// Nível de log
Logger.getLogger("org").setLevel(Level.ERROR)
Logger.getLogger("akka").setLevel(Level.ERROR)
//calcula o tempo de execucao do codigo
def time[R](block: => R): R = {
val t0 = System.nanoTime()
val result = block // call-by-name
val t1 = System.nanoTime()
println("Elapsed time: " + (t1 - t0).asInstanceOf[Double] / 1000000000.0 + "s")
result
}
// Variáveis
val spSession = ConfiguracaoSpark.sparkSess
val spContext = ConfiguracaoSpark.sparkCon
val datadir = ConfiguracaoSpark.datadir
time {
//Cria instancia da classe TratamentoCSV
```

```
val Tratar_dados = new TratamentoCSV
val Tratar_dados_RAT = new TratamentoCSV_RAT
//Cria dataframe com as estatisticas normalizadas
val Dados_conexoes = Tratar_dados.DF_CSV_Estatisticas_normalizado
val Dados_conexoes_RAT = Tratar_dados_RAT.DF_CSV_Estatisticas_normalizado
//cria variavel responsavel para atribuir o numero de clusters do Kmeans (K)
val Valor_K = Tratar_dados.servicos.toInt
val Valor_K_RAT = Tratar_dados_RAT.servicos.toInt

//metodo para executar clustering com o k-means - Agora so executa os modelos nos dados
def Clustering(cluster: String, dataframe: org.apache.spark.sql.DataFrame): org.apache.
spark.sql.DataFrame = {
var testData = dataframe
//carrega o modelo criado para aplicar em novos dados
var pipelineModel = PipelineModel.read.load(datadir + cluster)
var kmeansModel = pipelineModel.stages.last.asInstanceOf[KMeansModel]
var Dados_Teste = pipelineModel.transform(testData)
//testa a eficiencia do modelo criado com WSSSE
var WSSSE = kmeansModel.computeCost(Dados_Teste)
println(cluster)
println(s"Soma erro quadratico = $WSSSE")
var Resultado = Dados_Teste.drop("VetorCaracteristicas")
return Resultado
}
//Cria o dataframe com os dados de conexoes normais
var temp = Clustering("cluster" + 0, Dados_conexoes)
var temp_coluna = temp.select("Tempo_relativo", "cluster" + 0).withColumnRenamed(
"Tempo_relativo", "Tempo_relativo2")
var Resultado_clustering = Dados_conexoes.join(temp_coluna, Dados_conexoes(
"Tempo_relativo") === temp_coluna("Tempo_relativo2")).drop("Tempo_relativo2")
for (x <- 1 to 12) {
temp = Clustering("cluster" + x, Dados_conexoes)
temp_coluna = temp.select("Tempo_relativo", "cluster" + x).withColumnRenamed(
"Tempo_relativo", "Tempo_relativo2")
Resultado_clustering = Resultado_clustering.join(temp_coluna, Resultado_clustering(
"Tempo_relativo") === temp_coluna("Tempo_relativo2")).drop("Tempo_relativo2")
}
//Cria o dataframe com os dados de conexoes RAT
temp = Clustering("cluster" + 0, Dados_conexoes_RAT)
temp_coluna = temp.select("Tempo_relativo", "cluster" + 0).withColumnRenamed(
"Tempo_relativo", "Tempo_relativo2")
var Resultado_clustering_RAT = Dados_conexoes_RAT.join(temp_coluna,
Dados_conexoes_RAT(
"Tempo_relativo") === temp_coluna("Tempo_relativo2")).drop("Tempo_relativo2")
for (x <- 1 to 12) {
temp = Clustering("cluster" + x, Dados_conexoes_RAT)
temp_coluna = temp.select("Tempo_relativo", "cluster" + x).withColumnRenamed(
"Tempo_relativo", "Tempo_relativo2")
Resultado_clustering_RAT = Resultado_clustering_RAT.join(temp_coluna,
Resultado_clustering_RAT("Tempo_relativo") === temp_coluna("Tempo_relativo2")).drop(
"Tempo_relativo2")
```

```
}
/*
Random Forest
*/
//Cria coluna label com os valores 0 e 1
Resultado_clustering = Resultado_clustering.withColumn("label", lit(0))
// Resultado_clustering.show(5)
Resultado_clustering_RAT = Resultado_clustering_RAT.withColumn("label", lit(1))
// Resultado_clustering_RAT.show(5)
//Junta os dois dataframes
val Resultado_Final = Resultado_clustering.union(Resultado_clustering_RAT)
//Resultado_Final.agg(max("label")).show()
var caracteristicas = Resultado_Final.select("SEC","BEC","TB","BES/BEC", "Duracao",
"BEC/SEC","TS/Duracao","TB/Duracao","TS","BES","SES","SES/SEC","BES/SES","cluste
r0",
"cluster1","cluster2","cluster3","cluster4","cluster5","cluster6","cluster7","cluster8",
"cluster9","cluster10","cluster11","cluster12")
//Cria vetor com todas as caracteristicas
var assembler = new VectorAssembler().
setInputCols(caracteristicas.columns).
setOutputCol("features")
val Resultado_Final_caracteristicas = assembler.transform(Resultado_Final)
val model = RandomForestClassificationModel.load(datadir + "RandomForest")
var predictions = model.transform(Resultado_Final_caracteristicas)
predictions.select("Tempo_relativo", "Servidor", "S_Porta","Cliente", "C_Porta", "label",
"prediction").show(600)
predictions = predictions.select("prediction","label")
import spSession.implicits._
// var predictionsAndLabels = predictions.rdd.map {case Row(p: Double, l: Double) => (p,
l)}
var predictionsAndLabels = predictions.as[(Double,Double)].rdd
println("\nQuantidade por label:")
predictions.groupBy("label","prediction").count().show()
val metrics = new MulticlassMetrics(predictionsAndLabels)
println("Confusion Matrix:")
println(metrics.confusionMatrix.asML)
println("Accuracy:")
println(metrics.accuracy)
println()
val labels = metrics.labels
labels.foreach { l =>
println(s"Precision($l) = " + metrics.precision(l))
}
println()
// Recall por label
labels.foreach { l =>
println(s"Recall($l) = " + metrics.recall(l))
}
println()
// F-measure por label
```

```
labels.foreach { l =>
println(s"F1-measure($l) = " + metrics.fMeasure(l))
}
} }
```