UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

AGRUPAMENTO DE MALWARE POR COMPORTAMENTO DE EXECUÇÃO USANDO LÓGICA FUZZY

LINDEBERG PESSOA LEITE

ORIENTADOR: ANDRÉ RICARDO ABED GRÉGIO

COORIENTADOR: DANIEL GUERREIRO E SILVA

DISSERTAÇÃO DE MESTRADO EM ENGENHARIA ELÉTRICA
ÁREA DE CONCENTRAÇÃO INFORMÁTICA FORENSE E SEGURANÇA DA INFORMAÇÃO

PUBLICAÇÃO: PPGENE.DM - 639/2016

BRASÍLIA/DF: DEZEMBRO/2016.

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

AGRUPAMENTO DE MALWARE POR COMPORTAMENTO DE EXECUÇÃO USANDO LÓGICA FUZZY

LINDEBERG PESSOA LEITE

DISSERTAÇÃO DE MESTRADO PROFISSIONAL SUBMETIDA AO DEPARTAMENTO DE ENGENHARIA ELÉTRICA DA FACULDADE DE TECNOLOGIA DA UNIVERSIDADE DE BRASÍLIA, COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE.

APROVADA POR:

ANDRÉ RICARDO ABED GRÉGIO, Dr., UFPR
(ORIENTADOR)

FLÁVIO ELIAS GOMES DE DEUS, Dr., ENE/UNB
(EXAMINADOR INTERNO)

BRUNO WERNECK PINTO HOELZ, UNB
(EXAMINADOR EXTERNO)

Brasília, 27 de Dezembro de 2016.

FICHA CATALOGRÁFICA

LEITE, LINDEBERG PESSOA
Agrupamento de malware por comportamento
de execução usando lógica fuzzy. [Distrito Federal] 2016.
xxvii, 98 p., 201x297 mm (ENE/FT/UnB, Mestre, Engenharia Elétrica, 2016).
Dissertação de Mestrado - Universidade de Brasília.
Faculdade de Tecnologia. Departamento de Engenharia Elétrica.
1. Malware 2. Aprendizado de Máquina
3. Lógica Fuzzy
I. ENE/FT/UnB II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

Leite, L. P. (2016). Agrupamento de *malware* por comportamento de execução usando lógica *fuzzy*. Dissertação de Mestrado em Engenharia Elétrica, Publicação PPGENE.DM - 639/2016, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 98 p.

CESSÃO DE DIREITOS

NOME DO AUTOR: Lindeberg Pessoa Leite.

TÍTULO DA DISSERTAÇÃO DE MESTRADO: Agrupamento de *malware* por comportamento de execução usando lógica *fuzzy*

GRAU/ANO: Mestre/2016

Lindeberg Pessoa Leite
Rua Nigéria, nº 333, Torre 3, AP. 1802,
78.050-267 Cuiabá - MT - Brasil.

DEDICATÓRIA

À minha família, à minha noiva e aos meus amigos pelo apoio.

AGRADECIMENTOS

Ao Prof. Dr. André Ricardo Abed Grégio e ao Prof. Dr. Daniel Guerreiro e Silva pelo constante apoio, incentivo, dedicação e amizade essenciais para o desenvolvimento deste trabalho.

A todos os professores e colegas do Mestrado em Informática Forense, pela amizade e troca de experiência.

À Polícia Federal, por intermédio da Diretoria Técnico-Científica, e à Universidade de Brasília, por desenvolverem e apoiarem o projeto de Mestrado Profissional em Engenharia Elétrica com ênfase em Informática Forense e Segurança da Informação, no âmbito do qual esta pesquisa foi desenvolvida, e ao Ministério da Justiça, por fornecer os recursos financeiros necessários ao curso de Mestrado, por meio do Programa Nacional de Segurança Pública com Cidadania – PRONASCI.

Finalmente, agradeço aos meus pais, Lasaro Leite da Silva e Maria do Socorro Pessoa Leite, pelo incentivo incondicional e a minha noiva Marielly Matias Machado, pela compreensão do tempo que deixamos de aproveitar para que eu pudesse desenvolver atividades e realizar esta pesquisa.

Lindeberg Pessoa Leite

RESUMO

AGRUPAMENTO DE MALWARE POR COMPORTAMENTO DE EXECUÇÃO USANDO LÓGICA FUZZY

Autor: Lindeberg Pessoa Leite

Orientador: André Grégio
Coorientador: Daniel Guerreiro e Silva

Programa de Pós-graduação em Engenharia Elétrica

Brasília, dezembro de 2016

A ameaça de variantes de *malware* aumenta continuamente. Várias abordagens para agrupamento de *malware* já foram aplicadas para entender melhor como caracterizar suas famílias. Destas, a análise comportamental pode usar tanto métodos de aprendizado supervisionado como não supervisionado. Nesse último caso, a análise é comumente baseada em lógica convencional, em que um dado exemplar deve pertencer a apenas uma família. Neste trabalho, propõe-se uma abordagem de agrupamento comportamental por lógica *fuzzy*, que atribui um grau de relevância à cada exemplar e permite que este pertença a mais de uma família. Essa abordagem possibilita verificar outros comportamentos das amostras, não visualizados na lógica convencional. Compara-se o algoritmo escolhido — *Fuzzy C-Means (FCM)* — com o algoritmo *K-Means* para analisar similaridades e mostrar os benefícios do *FCM* na análise comportamental de *malware*. Nos resultados obtidos, verificou-se, nos casos em que há *clusters* sem rótulos, que o *FCM* apresentou vantagens na atribuição de um nome ao grupo devido à sua matriz de pertinência. Enquanto que no *K-Means* quatro *clusters* permaneceram sem rótulo, no caso do *FCM*, repetindo o processo, conseguiu-se atribuir rótulos a todos os *clusters*. Ademais, verificaram-se as semelhanças e divergências na aplicação de ambos os métodos e mensurou-se o tempo de execução dos experimentos. Por fim, conclui-se que outro benefício da aplicação de lógica *fuzzy* em relação ao método de lógica *crisp* reside no fato de que os programas maliciosos não se limitam apenas a um comportamento específico de uma dada família, ou seja, podem pertencer a várias delas ao mesmo tempo. Desse modo, a lógica *fuzzy* modela, de forma mais fidedigna, o real comportamento malicioso exibido durante uma infecção.

ABSTRACT

BEHAVIORAL MALWARE CLUSTERING USING FUZZY LOGIC

Author: Lindeberg Pessoa Leite

Supervisor: André Grégio

Programa de Pós-graduação em Engenharia Elétrica

Brasília, dezember of 2016

The threat of malware variants continuously increases. Several approaches have been applied to malware clustering for a better understanding on how to characterize families. Among them, behavioral analysis is one that can use supervised or unsupervised learning methods. This type of analysis is mainly based on conventional (crisp) logic, in which a particular sample must belong only to one malware family. In this work, we propose a behavioral clustering approach using fuzzy logic, which assigns a relevance degree to each sample and consequently enables it to be part of more than one family. This approach enables to check other behaviors of the samples, not visualized in conventional logic. We compare the chosen fuzzy logic algorithm — Fuzzy C-Means (FCM) — with K-Means so as to analyze their similarities and show the advantages of FCM for malware behavioral analysis. In the results, which there are clusters without labels, the FCM presented advantages in assigning a name to the group due to its relevance degree. While in the K-Means four clusters remained unlabeled, in the case of FCM, repeating the process, it was possible to assign labels to all clusters. In addition, the similarities and divergences of the methods were examined and their execution time of experiments were measured. Finally, it is concluded that another benefit of the application of fuzzy logic in relation to crisp logic method lies in the fact that malicious programs are not limited to a specific behavior of a given family, i.e, may be part of more than one family at the same time. Thus, fuzzy logic presents more reliably the actual malicious behavior reveal during an infection.

SUMÁRIO

LISTA DE TABELAS

LISTA DE FIGURAS

1 INTRODUÇÃO

1.1 MOTIVAÇÃO

A quantidade de *malware* vem aumentando, significativamente, ano após ano. De acordo com a McAfee (Intel Security, 2016), surgem 245 novas ameaças a cada minuto ou mais de quatro por segundo.

No segundo trimestre de 2016, já eram contabilizados mais de 40 milhões de novos tipos de *malware*. Esses exemplares, embora considerados "novos" ou "únicos", nada mais são do que mutações de *malware* já existentes. Pequenas variações no código, como mudanças em uma lista de endereços IP para varredura ou comunicação, em *strings* ou o uso de mecanismos de ofuscação (*packers*) fazem com que o exemplar seja considerado novo em relação aos demais. Os exemplares decorrentes de mutação de outros são conhecidos como *variantes*. As variantes, responsáveis pela elevação na quantidade de *malware* em atuação, dificultam a criação de assinaturas de detecção e facilitam a tarefa dos atacantes. Por outro lado, estimula-se a busca por técnicas capazes de detectar, classificar e/ou agrupar tais exemplares de maneira eficiente e eficaz.

Os fabricantes de antivírus (AV) criam assinaturas estáticas usando procedimentos majoritariamente manuais, pois é preciso que um analista humano descompile ou desmonte o código a fim de encontrar um padrão que, ao mesmo tempo em que detecte um *malware*, não gere falsos-positivos que prejudiquem a experiência do usuário. Por isso, grande parte dos exemplares de *malware*, incluindo os mais complexos, permanece não detectada por um tempo além do adequado. Por exemplo, estima-se um típico intervalo de 54 dias entre a disseminação de um dado *malware* e sua detecção por algum AV; 15% das amostras passam desapercebidas por 180 dias (DAMBALLA, 2009). Por isso, torna-se essencial desenvolver métodos automatizados para detectar e agrupar exemplares de *malware* desconhecidos, ou seja, que ainda não pertencem aos bancos de dados de assinaturas dos AVs (SALEHI; GHIASI; SAMI, 2012).

A dissecação de um exemplar de *malware* para melhor compreensão de seu modo de atuação (disseminação, infecção, ocultação de rastros, comunicação com o atacante,

entre outras ações) envolve formas distintas de análise: (*i*) a análise estática atua diretamente no binário sem a necessidade de execução, ou seja, obtêm-se informações do cabeçalho, verifica-se a presença de *packers*, descompila-se, quando possível, para se avaliar suas instruções e se tentar identificar blocos suspeitos, busca-se por *strings* que correspondam a endereços IP, URLs, e-mail etc.; (*ii*) a análise dinâmica consiste na observação da execução do *malware* em um ambiente controlado (*sandbox*) a fim de se extrair seu comportamento, isto é, quais são as atividades realizadas no nível do sistema operacional e da rede que podem ser utilizadas para definir como a infecção ocorre; (*iii*) a combinação de ambos os tipos, a fim de se valer das vantagens providas tanto pela análise estática quanto pela dinâmica.

Além da análise estática, dinâmica ou mista para detecção, há uma preocupação crescente com o agrupamento de exemplares na família correta. Com isso, entende-se melhor as características de cada grupo e aumenta-se a chance de detecção. Ademais, descobrir o rótulo adequado de um determinado artefato é fundamental para conhecer antecipadamente sua intenção maliciosa, bem como seu método de infecção e propagação. Em uma abordagem investigativa, saber o rótulo correto direciona o esforço do analista para as atividades maliciosas da amostra, inclusive para antever contramedidas. Nesse sentido, técnicas de aprendizado de máquina podem ser aplicadas para se realizar o processo de agrupamento (em inglês, *clustering*) de forma automatizada. Tradicionalmente, tais tentativas envolvem a aplicação de algoritmos fundamentados na lógica tradicional ou *"crisp"*, a qual se associa à teoria clássica de conjuntos: sob esta ótica, um exemplar de *malware* pertence exclusivamente a uma família, dentre todas as possíveis. No entanto, com a sofisticação dos processos de criação de *malware* usando kits de faça-você-mesmo, os exemplares produzidos passaram a empregar intensivamente o reuso de código. Dessa forma, o *malware* atual não é mais limitado a exibir o comportamento específico de uma classe pré-determinada, podendo carregar, ao mesmo tempo, características de várias classes. Logo, faz-se necessário um método capaz de retratar, de forma mais fidedigna, a dinâmica comportamental de um *malware*. Nesta dissertação, propõe-se um método para agrupamento de exemplares de *malware* pelo seu comportamento de execução, o qual tem sua base na lógica fuzzy. Realizou-se um estudo comparativo entre um método que emprega a lógica *crisp* — *K-Means* — e outro que adota a lógica fuzzy — *Fuzzy C-Means* (*FCM*) (BEZDEK; EHRLICH; FULL, 1984). Este trabalho, portanto, fornece as seguintes contribuições:

- aplicação de um novo método para agrupar exemplares de *malware* de forma mais flexível que os classificadores tradicionalmente utilizados;

- um processo de rotulação que considera os diversos comportamentos exibidos por um *malware* moderno. Nos casos em que há *clusters* sem rótulos, o *FCM* apresentou vantagens na atribuição de um nome ao grupo devido à sua matriz de pertinência;

- avaliação comparativa entre os algoritmos de agrupamento *K-Means* (lógica tradicional) e *FCM* (lógica *fuzzy*), permitindo a análise dos resultados para verificação de semelhanças entre eles, bem como a análise do tempo de execução gasto por algoritmos de diferentes paradigmas aplicados ao mesmo *dataset*. Constatou-se um alto grau de equivalência entre os *clusters* produzidos por ambos e mensurou-se o tempo de execução dos experimentos realizados com eles.

1.2 TRABALHOS RELACIONADOS

Nesta seção, são apresentados os trabalhos relacionados tanto envolvendo lógica convencional quanto lógica *fuzzy*. No que tange à lógica convencional, observa-se que o algoritmo *K-Means* é amplamente utilizado. Desse modo, os trabalhos relacionados envolvendo lógica convencional - principalmente o trabalho de (PIRSCOVEANU, 2015) - serviram de inspiração e, ao mesmo tempo, de comparação no desenvolvimento desta dissertação que emprega o algoritmo *Fuzzy C-Means*. Antes de apresentar os objetivos deste trabalho, vale destacar os principais trabalhos relacionados encontrados na literatura

(ANDRADE; MELLO; DUARTE, 2013) desenvolveram uma metodologia para análise automatizada de *malware* com classificação por aprendizado de máquina. Inicialmente, realizou-se o processo de coleta de programas, os quais foram divididos em dois conjuntos—*malware* e não *malware*. Os exemplares coletados foram enviados para execução na ferramenta de análise dinâmica Cuckoo Sandbox (GUARNIERI, 2016). As funções de API acessadas provenientes dos relatórios gerados pelo Cuckoo são comparadas com o conjunto de 121 funções de API comumente chamadas por programas maliciosos de acordo com (SIKORSKI; HONIG, 2012). O resultado dessa comparação produziu uma lista contendo as 20 funções de API mais relevantes para a detecção de atividades de *malware*. A essas 20 funções, os autores adicionaram mais dois atributos: o número de processos criados e o número de *downloads* realizados. Essa lista final composta por 22 atributos, denominada de dicionário de termos, foi transformada em um vetor de atributos no formato ARFF (*Attribute-Relation File Format*), para servir de entrada à ferramenta WEKA (HALL et al., 2009). O conjunto de dados

utilizados continha 66.463 amostras (maliciosas e benignas), as quais foram divididas em quatro experimentos para o processo de classificação. As técnicas que obtiveram melhor resultado foram o algoritmo J.48 (árvore de decisão) e o algoritmo *Random Forest*, ambos em implementações da ferramenta WEKA; este último apresentou uma acurácia (FAWCETT, 2006) de 93,4%, enquanto o outro apresentou acurácia de 92,2% para o primeiro experimento. Nos outros três experimentos, percentuais semelhantes foram obtidos.

(SALEHI; GHIASI; SAMI, 2012) criaram um método para detecção de *malware* baseado nas chamadas de função de API e seus argumentos. Primeiramente, efetuou-se o processo de coleta de exemplares, os quais foram divididos em dois conjuntos (*malware* e não *malware*), nos quais 385 deles eram programas nativos encontrados no sistema operacional Windows ou outras ferramentas consideradas benignas e 826 eram programas considerados maliciosos. Esses arquivos foram obtidos de (SAMI et al., 2010). Os binários executaram em um sistema operacional *Windows* virtualizado *VMware* e o comportamento foi monitorado pelo `WINAPIOverride32` [1]. Foram monitoradas 126 funções de API das seis DLLs (*Dynamic-Link Library*) mais importantes do Windows: `advapi32.dll`, `kernel32.dll`, `ntdll.dll`, `user32.dll`, `wininet.dll` e `ws2_32.dll`. Além da monitoração das funções de API, foram observados também seus argumentos. O resultado dessa monitoração foi convertido para o formato ARFF e, no processo de classificação, a melhor acurácia (97%) foi obtida com o algoritmo *Functional Trees* (FT) do WEKA.

(MANGIALARDO, 2015) elaborou uma técnica integrada de análises estática e dinâmica na identificação de *malware* utilizando aprendizado de máquina. O processo iniciou-se com a coleta de exemplares, então divididos em dois conjuntos: um composto por 2.659 executáveis benignos obtidos do *SourceForge*[2], *OldApps*[3] e outras fontes; outro por 131.073 exemplares de *malware* obtidos do *VirusShare*[4]. No processo de escolha das características, levou-se em consideração tanto *strings* extraídas da análise estática como funções de API capturadas da análise dinâmica. Para formar o dicionário de *strings* da análise estática, usaram-se várias ferramentas para extrair as informações

[1] WinAPIOverride é um software avançado de monitoramento de API para programas de 32 e 64 bits.

[2] `https://sourceforge.net`

[3] `http://www.oldapps.com`

[4] `https://virusshare.com`

mais relevantes, tais como *Yara*[5], *PEiD*[6] e dados obtidos pelo *PEScanner*[7]. No caso da análise dinâmica, montou-se um dicionário de termos semelhante ao visto em (ANDRADE; MELLO; DUARTE, 2013). Para o processo de aprendizado, foi utilizado o framework FAMA (DUARTE et al., 2012). Os algoritmos de aprendizado de máquina escolhidos para a tarefa de classificação foram o C5.0 (árvore de decisão) e a técnica *Random Forest*, cuja acurácia foi de 95,75%.

(PROVATAKI; KATOS, 2013) apresentaram um *framework* para análise forense de *malware* baseado em Cuckoo Sandbox, usado para avaliar e elaborar relatórios sobre o comportamento dos exemplares executados. O trabalho abordou as limitações da análise dinâmica e ampliou a funcionalidade da ferramenta Cuckoo Sandbox a fim de automatizar o processo de correlacionamento e investigação de várias execuções de um binário suspeito sobre plataformas distintas de sistemas operacionais. A abordagem apresentada permite que o analista identifique mudanças comportamentais quando o artefato é executado e possibilita responder questões relacionadas às atividades do programa malicioso que auxiliem uma investigação mais aprofundada.

(FIRDAUSI et al., 2010) demonstraram uma prova de conceito de um método para detecção de *malware*. Inicialmente, o comportamento dos artefatos foi obtido via análise dinâmica no sistema Anubis (ISECLAB, 2015), e os relatórios gerados foram pré-processados de forma a se criar vetores para classificação. Procedeu-se uma comparação de desempenho de cinco diferentes técnicas — *K-Nearest Neighbors* (KNN), *Naive Bayes*, Árvore de Decisão (J.48), *Support Vector Machine* (SVM) e Rede Neural do tipo Perceptron de Múltiplas Camadas (MLP). Foi produzido um pequeno conjunto de amostras maliciosas e dados de executáveis benignos. Os resultados obtidos revelaram que, em geral, o melhor desempenho é conseguido pela árvore de decisão J.48 com uma taxa de falsos-positivos de 2,4% e acurácia de 96,8%.

(PIRSCOVEANU, 2015) investigou as inconsistências associadas à geração de rótulos por mecanismos antivírus no processo de aprendizado não supervisionado, com objetivo de realizar análise comportamental de *malware*. Uma versão personalizada do Cuckoo Sandbox foi utilizada para coletar ações de cerca de 270.000 exemplares maliciosos e, posteriormente, criar um dicionário de termos consistindo de chamadas de API que foram executadas com sucesso e aquelas que falharam, com seus respectivos códigos de retorno. Além disso, avaliaram-se os resultados de detecção fornecidos por fabricantes

[5] `http://virustotal.github.io/yara/`

[6] `https://www.aldeid.com/wiki/PEiD`

[7] `https://code.google.com/archive/p/malwarecookbook/`

de antivírus. Com isso, criou-se uma solução que estabelece como rótulo o nome de família que recebeu a maioria do votos dos AVs. Devido às inconsistências entre os rótulo de AVs, a distância de Levensthein foi usada a fim de medir a similaridades entre os rótulos.

(HUANG et al., 2013) exploram a possibilidade de utilizar técnicas *fuzzy* para identificar semelhanças entre famílias de softwares maliciosos. No modelo proposto, características maliciosas de um programa são inicialmente captadas pelo reconhecimento de padrão *fuzzy* e, em seguida, uma ontologia *fuzzy* para análise comportamental de *malware* é apresentada. A abordagem é composta por uma inferência *fuzzy* e um mecanismo semântico de tomada de decisão para descrever as mudanças do valor de chave de registo, conexão de rede e alteração de arquivos, permitindo construir a ontologia e regras de comportamento. O modelo é capaz de detectar programas maliciosos desconhecidos e variantes de *malware*.

(ZHANG et al., 2010) propõem um método para detectar códigos maliciosos desconhecidos por meio de rede neural *fuzzy*. O sistema desenvolvido é composto por três módulos: *File-decompile*, *Behavior-recognize* e *Neural-network*. No primeiro, realiza-se uma decompilação do arquivo; no segundo, captura-se o comportamento do programa; no último, aplica-se o método de redes neurais. Neste, há um sistema de detecção de malware baseados em inferências *fuzzy* que tenta detectar se o comportamento é malicioso. Basicamente, o fluxo inicia-se com a extração da sequência de instruções e das chamadas de API para compor o comportamento do programa. Esses dados servem de entrada para o sistema de detecção. Nesse sistema, baseado na lógica *fuzzy*, infere-se um grau de maliciosidade da amostra, permitindo que esta seja classificada em três conjuntos: grupos maliciosos, grupo suspeitos e grupo benigno. Os resultados experimentais provam que a abordagem é eficaz na detecção de códigos maliciosos. No entanto, há várias técnicas de ofuscação que o sistema proposto não abrange.

Tendo-se em vista a quantidade elevada de pesquisas utilizando lógica convencional frente às da lógica *fuzzy*, resolveu-se avaliar os resultados deste trabalho com os produzidos pela lógica convencional, para indicar limitações dessa última. Mais especificamente, ao analisar a tese de (PIRSCOVEANU, 2015), baseada em lógica convencional, propõe-se uma melhoria que possibilita rotular clusters desconhecidos. Ademais, também se realizou uma análise comparativa entre o algoritmo de agrupamento *K-Means*, bastante empregado na literatura, e o algoritmo *FCM*.

1.3 OBJETIVO

Este trabalho tem como principal objetivo desenvolver um método para agrupar exemplares de *malware* baseado na lógica *fuzzy*, facilitando rotular grupos nos casos que os artefatos ainda não tenham sido rotulados pelos fabricantes de antivírus. Ademais, ao utilizar lógica *fuzzy*, objetiva-se obter mais informações acerca do *malware* por meio da matriz de pertinência. Para comprovar isso, realiza-se um estudo comparativo entre o principal algoritmo usada para agrupar *malware* - K-Means - com o FCM, a fim de se identificar as vantagens de utilizar este último. Por fim, nesse estudo, verifica-se também as semelhanças e diferenças entre *clusters* gerados por ambos os métodos, apresentando uma justificativa para os casos divergentes.

1.4 ORGANIZAÇÃO DA DISSERTAÇÃO

Esta dissertação está organizada da seguinte forma: o capítulo 2 apresenta uma visão geral das diferentes classes de programas maliciosos, bem com discute a análise de *malware*, ao descrever as as duas principais técnicas de análises: estática e dinâmica. No capítulo 3, são detalhados os métodos de aprendizado não supervisionado (K-Means e FCM) usados neste trabalho, bem como processos de validação. No capítulo 4, apresentam-se as fases do método proposto: coleta de exemplares de *malware*, captura das de funções das APIs e geração do dicionário de termos. Além disso, realiza-se uma redução de dimensionalidade por meio do PCA e executam-se o K-Means e FCM. No capítulo 5, os resultados obtidos por meio da aplicação dos dois algoritmos de aprendizado de máquina são analisados sob três aspectos: semelhança entre *clusters* produzidos, processo de rotulação e tempo de execução e escalabilidade. No último capítulo, são debatidos os trabalhos futuros e conclusão.

2 ANÁLISE DE MALWARE

2.1 INTRODUÇÃO

A análise de *malware* consiste em identificar códigos maliciosos. Compreende atividades para a identificação de suas características baseando-se na forma como são obtidos, como são instalados, como se propagam, e como atuam nos sistemas. O *malware* é um software desenvolvido com o objetivo de causar danos a um sistema de computador que pode ou não estar conectado a uma rede de computadores. Sua motivação pode ser a obtenção de ganhos ilícitos, causar prejuízos a uma organização, a uma pessoa, ao funcionamento de infraestruturas críticas (MANGIALARDO, 2015). Os códigos maliciosos possuem várias formas de ataque, a depender do grupo ao qual pertença. Desse modo, as classes desses artefatos devem ser conhecidas para que se possa entender melhor suas características.

2.2 CLASSIFICAÇÃO DE MALWARE

Esta seção apresenta uma visão geral das diferentes classes de programas maliciosos. Essas classes não são mutuamente exclusivas, isto é, instâncias de malware podem apresentar características de várias classes ao mesmo tempo. Uma discussão mais detalhada de códigos maliciosos pode ser encontrado em Szor (SZOR, 2005) e em CERT.BR (CERT, 2016). Szor classifica os malware nas classes *vírus, worm, trojan, backdoor, downloader, dropper e rootkit.* O CERT.BR classifica e descreve nos seguintes grupos:

Vírus é um programa malicioso que se propaga inserindo cópias de si mesmo e se tornando parte de outros programas e arquivos.

O vírus depende da execução do programa ou arquivo hospedeiro a fim de se tornar ativo e dar continuidade ao processo de infecção, ou seja, para que o seu computador se infecte é necessário que um programa já infectado seja executado. Na atualidade,

as mídias removíveis tornaram-se o principal meio de propagação, sobretudo os *pen drives*.

Existem vários tipos de vírus. Alguns ficam ocultos, infectando arquivos do disco e executando uma série de atividades sem o conhecimento do usuário. Há outros que permanecem inativos durante certos períodos, entrando em atividade apenas em datas específicas. Alguns dos tipos de vírus mais comuns são:

- vírus propagado por *e-mail*: recebido como um arquivo anexo a um e-mail cujo conteúdo tenta induzir o usuário a clicar sobre este arquivo, fazendo com que seja executado;
- vírus de script: escrito em linguagem de script, como VBScript e JavaScript, e recebido ao acessar uma página Web ou por *e-mail*, como um arquivo anexo ou como parte do próprio e-mail escrito em formato HTML;
- vírus de macro: tipo específico de vírus de script, escrito em linguagem de macro, que tenta infectar arquivos manipulados por aplicativos que utilizam esta linguagem. Por exemplo, os que compõe o Microsoft Office (Excel, Word e PowerPoint, entre outros).

Worm é um programa capaz de se propagar, automaticamente, pelas redes, enviando cópias de si mesmo de computador para computador.

Diferentemente do vírus, o *worm* não se propaga por meio da inclusão de cópias de si mesmo em outros programas ou arquivos, mas sim pela execução direta de suas cópias ou pela exploração automática de vulnerabilidades existentes em programas instalados em computadores.

Worms são, notadamente, responsáveis por consumir muitos recursos, devido à grande quantidade de cópias de si mesmo que costumam propagar e, como consequência, podem afetar o desempenho de redes e a utilização de computadores.

O processo de propagação e infecção dos worms ocorre da seguinte maneira:

- identificação dos computadores alvos: após infectar um computador, o *worm* tenta se propagar e continuar o processo de infecção. Para isto, necessita identificar os computadores alvos para os quais tentará se copiar, o que pode ser

feito de uma ou mais das seguintes maneiras: efetuar varredura na rede e identificar computadores ativos, aguardar que outros computadores contatem o computador infectado, utilizar listas predefinidas ou obtidas na Internet contendo a identificação dos alvos ou utilizar informações contidas no computador infectado, como arquivos de configuração e listas de endereços de e-mail;

- envio das cópias: após identificar os alvos, o worm efetua cópias de si mesmo e tenta enviá-las para estes computadores, por uma ou mais das seguintes formas: como parte da exploração de vulnerabilidades existentes em programas instalados no computador- alvo, anexadas a *e-mails*, via canais de IRC (*Internet Relay Chat*), via programas de troca de mensagens instantâneas ou se incluindo em pastas compartilhadas em redes locais ou do tipo *P2P (Peer to Peer)*;

- ativação das cópias: após realizado o envio da cópia, o *worm* necessita ser executado para que a infecção ocorra, o que pode acontecer de uma ou mais das seguintes maneiras: pela exploração de vulnerabilidades em programas sendo executados no computador alvo no momento do recebimento da cópia, diretamente pelo usuário, ou pela realização de uma ação específica do usuário, a qual o *worm* está condicionado, como, por exemplo, a inserção de uma mídia removível;

- reinício do processo: após o alvo ser infectado, o processo de propagação e infecção recomeça, e o computador que antes era o alvo passa a ser também o computador originador dos ataques.

Bot e botnet

Bot é um programa que dispõe de mecanismos de comunicação com o invasor que permitem que ele seja controlado remotamente. Possui processo de infecção e propagação similar ao do worm, ou seja, é capaz de se propagar automaticamente, explorando vulnerabilidades existentes em programas instalados em computadores.

A comunicação entre o invasor e o computador infectado pelo bot pode ocorrer via canais de IRC, servidores Web e redes do tipo P2P, entre outros meios. Ao se comunicar, o invasor pode enviar instruções para que ações maliciosas sejam executadas, como desferir ataques, furtar dados do computador infectado e enviar spam.

Um computador infectado por um bot costuma ser chamado de zumbi (zombie computer), pois pode ser controlado remotamente, sem o conhecimento do seu dono.

Botnet é uma rede formada por centenas ou milhares de computadores zumbis e que permite potencializar as ações danosas executadas pelos bots.

Quanto mais zumbis participarem da botnet mais potente ela será. O atacante que a controlar, além de usá-la para seus próprios ataques, também pode alugá-la para outras pessoas ou grupos que desejem que uma ação maliciosa específica seja executada.

Algumas das ações maliciosas que costumam ser executadas por intermédio de botnets são: ataques de negação de serviço, propagação de códigos maliciosos (inclusive do próprio bot), coleta de informações de um grande número de computadores, envio de spam e camuflagem da identidade do atacante (com o uso de proxies instalados nos zumbis).

Spyware é um programa projetado para monitorar as atividades de um sistema e enviar as informações coletadas para terceiros.

Pode ser utilizado tanto de forma legítima quanto maliciosa, dependendo de como é instalado, das ações realizadas, do tipo de informação monitorada e do uso que é feito por quem recebe as informações coletadas. No uso legítimo, é instalado em um computador pessoal, pelo próprio dono ou com consentimento deste, com o objetivo de verificar se outras pessoas o estão utilizando de modo abusivo ou não autorizado. No uso malicioso, executa ações que podem comprometer a privacidade do usuário e a segurança do computador, como monitorar e capturar informações referentes à navegação do usuário ou inseridas em outros programas (por exemplo, conta de usuário e senha).

Alguns tipos específicos de programas spyware são:

- keylogger: capaz de capturar e armazenar as teclas digitadas pelo usuário no teclado do computador. Sua ativação, em muitos casos, é condicionada a uma ação prévia do usuário, como o acesso a um site específico de comércio eletrônico ou de Internet Banking;
- screenlogger: similar ao keylogger, capaz de armazenar a posição do cursor e a tela apresentada no monitor, nos momentos em que o mouse é clicado, ou a região que circunda a posição onde o mouse é clicado. Esse tipo é bastante utilizado por

atacantes para capturar as teclas digitadas pelos usuários em teclados virtuais, disponíveis principalmente em sites de Internet Banking;

- adware: projetado especificamente para apresentar propagandas; pode ser usado para fins legítimos, quando incorporado a programas e serviços, como forma de patrocínio ou retorno financeiro para quem desenvolve programas livres ou presta serviços gratuitos. Também pode ser usado para fins maliciosos, quando as propagandas apresentadas são direcionadas, de acordo com a navegação do usuário e sem que este saiba que tal monitoramento está sendo feito.

Backdoor é um programa que permite o retorno de um invasor a um computador comprometido, por meio da inclusão de serviços criados ou modificados para esse fim.

Pode ser incluído pela ação de outros códigos maliciosos, que tenham previamente infectado o computador, ou por atacantes, que exploram vulnerabilidades existentes nos programas instalados no computador para invadi-lo.

Após incluído, o backdoor é usado para assegurar o acesso futuro ao computador comprometido, permitindo que ele seja acessado remotamente, sem que haja necessidade de recorrer novamente aos métodos utilizados na realização da invasão ou infecção e, na maioria dos casos, sem que seja notado.

A forma usual de inclusão de um backdoor consiste na disponibilização de um novo serviço ou na substituição de um determinado serviço por uma versão alterada, normalmente possuindo recursos que permitem o acesso remoto. Programas de administração remota, como BackOrifice, NetBus, SubSeven, VNC e Radmin, se mal configurados ou utilizados sem o consentimento do usuário, também podem ser classificados como backdoors.

Há casos de backdoors incluídos, propositalmente, por fabricantes de programas, sob alegação de necessidades administrativas. Esses casos constituem uma séria ameaça à segurança de um computador que contenha um destes programas instalados pois, além de comprometerem a privacidade do usuário, também podem ser usados por invasores para acessarem remotamente o computador.

Cavalo de troia (Trojan)

Cavalo de troia, trojan ou trojan-horse, é um programa que, além de executar as funções para as quais foi aparentemente projetado, também executa outras funções, normalmente maliciosas e sem o conhecimento do usuário.

Há diferentes tipos de trojans, classificados de acordo com as ações maliciosas que costumam executar ao infectar um computador. Alguns desses tipos são:

- *Trojan Downloader*: instala outros códigos maliciosos, obtidos de sites na Internet;
- *Trojan Dropper*: instala outros códigos maliciosos, embutidos no próprio código do *trojan*;
- *Trojan Backdoor*: inclui *backdoors*, possibilitando o acesso remoto do atacante ao computador;
- *Trojan DoS*: instala ferramentas de negação de serviço e as utiliza para desferir ataques;
- *Trojan* Destrutivo: altera/apaga arquivos e diretórios, formata o disco rígido e pode deixar o computador fora de operação;
- *Trojan Clicker*: redireciona a navegação do usuário para sites específicos, com o objetivo de aumentar a quantidade de acessos a estes sites ou apresentar propagandas;
- *Trojan Proxy*: instala um servidor de *proxy*, possibilitando que o computador seja utilizado para navegação anônima e para envio de spam;
- *Trojan Spy*: instala programas spyware e os utiliza para coletar informações sensíveis, como senhas e números de cartão de crédito, e enviá-las ao atacante;
- *Trojan Banker*: coleta dados bancários do usuário, por meio da instalação de programas spyware que são ativados quando sites de Internet *Banking* são acessados. É similar ao *Trojan Spy* porém com objetivos mais específicos.

Rootkit é um conjunto de programas e técnicas que permite esconder e assegurar a presença de um invasor ou de outro código malicioso em um computador comprometido.

O conjunto de programas e técnicas fornecido pelos rootkits pode ser usado para:

- remover evidências em arquivos de logs;
- instalar outros códigos maliciosos, como backdoors, para assegurar o acesso futuro ao computador infectado;
- esconder atividades e informações, como arquivos, diretórios, processos, chaves de registro, conexões de rede etc;
- mapear potenciais vulnerabilidades em outros computadores, por meio de varreduras na rede;
- capturar informações da rede onde o computador comprometido está localizado, pela interceptação de tráfego.

É muito importante ressaltar que o nome rootkit não indica que os programas e as técnicas que o compõem são usados para obter acesso privilegiado a um computador, mas sim para mantê-lo.

Rootkits inicialmente eram usados por atacantes que, após invadirem um computador, os instalavam para manter o acesso privilegiado, sem precisar recorrer novamente aos métodos utilizados na invasão, e para esconder suas atividades do responsável e/ou dos usuários do computador. Apesar de, ainda, serem bastante usados por atacantes, os rootkits atualmente têm sido também utilizados e incorporados por outros códigos maliciosos para ficarem ocultos e não serem detectados pelo usuário e nem por mecanismos de proteção.

2.3 TÉCNICAS DE ANÁLISE DE MALWARE

De acordo com (MALIN; AQUILINA; CASEY, 2011), o processo de identificação de um artefato malicioso consiste nas seguintes etapas: preservação forense e análise dos dados voláteis; exame de memória do sistema infectado; análise forense dos meios de armazenamento permanentes; exame das características do arquivo desconhecido; e análise estática e dinâmica do arquivo suspeito.

(MALIN; AQUILINA; CASEY, 2011) esclarecem que a etapa 1 compreende a preservação do estado dos dados voláteis, ou seja, aqueles que estão disponíveis apenas enquanto o sistema estiver ligado. Vale destacar que os dados voláteis não se referem

somente aos dados em memória principal, mas também refere-se a informações como endereços IP, registros de eventos de segurança em logs do sistema, além de outros detalhes que juntos podem fornecer um completo entendimento do *malware*. A etapa 2 corresponde à captura de dumps de memória. Nela, obtêm-se informações acerca do malware e vestígios de sua atuação. Para isso, necessita de software que auxilie no processo de verificação devido ao grande volume de estruturas de dados presentes nos dumps de memória. O software *Volatility* [8] é bastante empregado para essa análise. Trata-se de ferramenta forense para resposta a incidentes e análise de *malware*. Na etapa 3, o especialista busca encontrar vestígios da presença do *malware* em locais como o disco rígido, incluindo os arquivos armazenados, arquivos de registros e logs, entre outros. Essas informações podem ser úteis para identificar, por exemplo, a fonte do ataque e a funcionalidade do *malware*. Na etapa 4, inicia-se a análise do arquivo suspeito. Busca-se entender as funções do arquivo, como ele pode ser caracterizado, por exemplo, como benigno ou maligno, se o arquivo possui conteúdo ofuscado, se sua origem é desconhecida, se é um arquivo conhecido armazenado em local pouco usual, se produz atividade de rede anormal. Esta fase emprega uma análise estática inicial do código binário para obter informações como dependências do arquivo, assinaturas de antivírus, metadados associados com o arquivo suspeito. Na etapa 5, as duas técnicas de análise de *malware*, isto é, a análise estática e a dinâmica são utilizadas, possibilitando que o analista faça a classificação do arquivo suspeito.

O processo de análise de um determinado programa durante sua execução é chamado de análise dinâmica, enquanto a análise do *programa* sem executá-lo consiste na análise estática. A análise dinâmica é detalhada na Seção 2.3.7. Na próxima seção, descreve-se a análise estática.

2.3.1 ANÁLISE ESTÁTICA

A análise estática descreve o processo de analisar o código ou estrutura de um programa para determinar sua função. Fundamenta-se no exame de um *sofware* sem executá-lo. Nessa análise, observam-se padrões em suas *strings*, sequência de *bytes*,chamadas a bibliotecas estaticamente vinculadas, grafos de controle de fluxo (CFG), distribuição de *opcodes*, entre outros (GANDOTRA; BANSAL; SOFAT, 2014).

Nesta seção, baseada em (SIKORSKI; HONIG, 2012), discutem-se várias maneiras de

[8] http://www.volatilityfoundation.org/

extrair informações úteis de executáveis, por meio das seguintes técnicas:

- usando ferramentas antivírus para confirmar se possui atividade maliciosa;
- usando *hashes* para identificar *malware*;
- recuperando informação de strings, funções e cabeçalhos de arquivo.

As técnicas utilizadas dependem dos objetivos. Normalmente, empregam-nas em conjunto para reunir o máximo de informação possível. Para iniciar a análise de um *malware*, um bom primeiro passo é executá-lo por meio de vários programas AV, que já podem tê-lo identificado antes. No entanto, as ferramentas de AV não são perfeitas, uma vez que essas se baseiam principalmente em banco de dados de código suspeito conhecido (assinaturas de arquivos), bem como comportamentais e de correspondência de padrões (heurística) para identificar arquivos suspeitos. Um problema é que os fabricantes de *malware* podem facilmente modificar seu código, mudando assim a assinatura de seu programa e escapar de *scanners* de vírus. Além disso, *malware* mais raro, muitas vezes, passa despercebido por um software antivírus, porque simplesmente ainda não se encontra na base de dados. No caso da heurística, embora muitas vezes bem sucedida em identificar código malicioso desconhecido, pode ser contornada por meio de um novo artefato. Nesse sentido, como os AV usam diferentes assinaturas e heurísticas, uma abordagem interessante é executar vários AV contra a mesma amostra. O site VirusTotal tem sido amplamente empregado, uma vez que permite submeter um *malware* a vários AV ao mesmo tempo. Já o *Hashing* é um método comumente usado para identificar *malware*. O software malicioso é executado por meio de um programa de *hash* que produz um valor único (hash), que identifica o *malware* univocamente (uma espécie de impressão digital). O *Message-Digest Algorithm* 5 (MD5) é frequentemente utilizado para análise de *malware*. Por fim, a pesquisa por meio de *strings* no programa pode ser uma maneira simples de se obter informações acerca da sua funcionalidade. Por exemplo, se o programa acessa uma URL, esta é armazenada como uma *string* nele.

2.3.2 *PACKING* E OFUSCAÇÃO DE *MALWARE*

Os fabricantes de *malware* utilizam, frequentemente, de *packing* ou ofuscação para fazer seus arquivos mais difíceis de detectar ou analisar. Programas ofuscados são aqueles

cuja execução o autor do artefato tenta esconder, enquanto programas que sofreram *packing* são um subconjunto de programas ofuscados em que o programa malicioso é comprimido e não pode ser analisado. Ambas as técnicas limitam tentativas de análise estática. Programas que não passaram por essas técnicas quase sempre possuem muitas *strings*. *Malware* ofuscado ou que sofrer *packing* contém poucas *strings*. Se um programa contiver poucas *strings*, é provavelmente que ele esteja ofuscado ou sofrido *packing*. Provavelmente, faz-se necessário utilizar outras técnicas, além da análise estática.

2.3.3 *PORTABLE EXECUTABLE (PE)* E FORMATO DE ARQUIVOS

O formato de um arquivo pode revelar muito sobre a funcionalidade do programa. O formato de arquivo *Portable Executable (PE)* é uma estrutura de dados que contêm as informações necessárias para o carregador do *Windows* gerenciar o código executável. Quase todos os arquivos com código executável que são carregado pelo *Windows* estão no formato de arquivo PE, embora alguns formatos de arquivo legados aparecem em raras ocasiões na análise de *malware*. Arquivos PE começam com um cabeçalho que inclui informações sobre o código, o tipo de aplicação, funções de biblioteca e espaço requeridos. As informações no cabeçalho PE são de grande valor para o analista de *malware*.

2.3.4 FUNÇÕES E BIBLIOTECAS

Uma das informações mais úteis que se pode reunir acerca de um executável é a lista de funções que ele importa. As importações são funções usadas por um programa que estão armazenadas em um *software* diferente, tal como acontece com o código de bibliotecas que contêm funcionalidade comum a muitos programas. Código de bibliotecas podem ser conectados ao executável principal por meio do *linking*. Programadores fazem importações para os seus programas para que eles não precisem reimplementar certas funcionalidades em vários programas. Bibliotecas podem ser vinculadas estaticamente, em tempo de execução, ou dinamicamente. Saber como o código da biblioteca está vinculado é fundamental para compreensão do *malware*, uma vez que a informação encontrada no cabeçalho do arquivo PE depende de como o código da biblioteca foi vinculada.

2.3.5 VINCULAÇÃO ESTÁTICA, DINÂMICA E EM TEMPO DE EXECUÇÃO

Vinculação estática é o método menos comumente usado para vincular bibliotecas, embora seja mais utilizado em programas para *UNIX* ou *Linux*. Quando uma biblioteca é estaticamente vinculada a um executável, todo seu código é copiado para o executável, que aumenta de tamanho. Ao analisar código, é difícil diferenciar entre o código estaticamente vinculado e o próprio código do executável, porque nada no cabeçalho do arquivo PE indica que o arquivo contém código vinculado. Embora impopular em programas não maliciosos, vínculo em tempo de execução (*runtime*) é frequentemente usado em *malware*, especialmente quando é ofuscado ou passou por um *packer*. Executáveis que usam *runtime* vinculam as bibliotecas somente quando essa função é necessária.

Várias funções do *Windows* permitem que os programadores importem funções não constantes no cabeçalho do arquivo de um programa. Dessas, as duas mais comumente utilizadas são *LoadLibrary* e *GetProcAddress*, as quais possibilitam que um *software* acesse qualquer função nas bibliotecas no sistema, o que significa que, quando essas funções são usadas, não se pode dizer estaticamente quais funções estão sendo vinculadas pelo programa suspeito.

De todos os métodos, o vínculo dinâmica é o mais comum e interessante para analistas de artefatos. Diferentemente do vínculo em tempo de execução, as bibliotecas são ligadas no início do programa. Quando as bibliotecas estão vinculadas dinamicamente, o sistema operacional procura bibliotecas necessárias quando o programa é carregado. Nesse caso, o programa chama a função de biblioteca vinculada, e ela é executada dentro da biblioteca.

O cabeçalho de um arquivo *PE* guarda informações sobre cada biblioteca que será carregada e todas as funções que serão utilizados pelo programa. As bibliotecas empregadas e as funções chamadas, muitas vezes, são as partes mais importantes de um programa, e identificá-las é particularmente importante, porque permite inferir o que o programa faz. Por exemplo, se um programa importa a função *URLDownloadToFile*, imagina-se que ele se conecta à Internet para baixar algum arquivo que, em seguida, é armazenado em algum local.

2.3.6 CABEÇALHO E SEÇÕES DO PE

Cabeçalho de arquivo PE pode fornecer muito mais informações do que apenas as importações. O formato do PE contém um cabeçalho seguido por uma série de secções. O cabeçalho contém metadados sobre o próprio arquivo. Após o cabeçalho, estão as secções reais do arquivo, cada um com informação útil. A seguir são descritas as principais seções em um arquivo PE:

- .text - Contém as instruções que a CPU executa. Todas as outras seções armazena dados e informações de apoio. Geralmente, esta é a única seção que pode executar e deve ser a única que inclui código.
- .rdata - Normalmente, contém informações de importação e exportação. Esta seção também pode armazenar outros dados de somente leitura usados pelo programa. Às vezes, um arquivo pode conter uma .idata e seção de .edata, que armazenam as informações de importação e exportação, respectivamente.
- .data - Possui dados globais do programa, que é acessível a partir de qualquer lugar no arquivo. Dados locais não são armazenados nesta seção, ou em qualquer outro lugar no arquivo PE.
- .rsrc - Inclui os recursos utilizados pelo executável que não são considerados parte do programa, como ícones, imagens, menus, e *strings*. Estas últimas podem ser armazenadas na seção .rsrc ou no programa principal, mas são frequentemente armazenadas na seção .rsrc para suportar vários idiomas.

O cabeçalho PE contém informações úteis para o analista de *malware*. As principais informações que podem ser obtidas a partir de um cabeçalho de PE são:

- *Imports* - Funções de outras bibliotecas usadas pelo *malware*.
- *Exports* - Funções no artefato que se destinam a ser chamado por outros programas ou bibliotecas.
- *Time Date Stamp* - Data em que o programa foi compilado.
- *Sections* - Nomes de seções no arquivo e seus tamanhos em disco e na memória.
- *Subsystem* - Indica se o programa é uma linha de comando ou aplicativo GUI.

- *Resources* - *Strings*, ícones, menus e outras informações incluídas no arquivo

Por fim, para a análise ser realizada com sucesso, o executável precisa estar desempacotado e descriptografado, pois técnicas de ofuscação como criptografia, compressão, inserção de código e permutação não podem ser facilmente detectadas via método normal de análise estática (HU, 2011). Desse modo, uma análise estática aplicada a um exemplar de *malware* ofuscado pode se transformar em um problema NP-difícil (MOSER; KRUEGEL; KIRDA, 2007). Ademais, geralmente, o código-fonte do exemplar não está prontamente disponível. Isso reduz a análise estática a recuperação das informações do *malware* na representação binária. Analisar binários traz desafios. Por exemplo, a desmontagem de tais programas podem gerar resultados ambíguos se o binário emprega técnicas de código de automodificação. Além disso, há valores que não podem ser determinados estaticamente (por exemplo, data atual do sistema, instruções de salto indiretos) e agravam a aplicação de técnicas de análise estática. Nesse sentido, pode-se esperar que os autores de *malware*, ao saber dessas limitações, criem instâncias de artefatos que empregam estas técnicas para frustrar a análise estática. Portanto, é necessário desenvolver técnicas de análise que sejam resistentes a tais modificações e possibilitem uma análise mais completa (EGELE et al., 2012).

2.3.7 ANÁLISE DINÂMICA

A análise dinâmica é qualquer exame realizado após a execução de um *malware*. Pode envolver o monitoramento de artefato enquanto ele é executado ou exame do sistema após a execução do exemplar. Ao contrário de análise estática, análise dinâmica permite observar a verdadeira funcionalidade do *malware*, porque, por exemplo, a existência de uma *string* sugerindo uma ação no binário não significa que a ação vai realmente ser executada. A análise dinâmica é também uma forma eficiente para identificar a funcionalidade de uma artefato. Por exemplo, se o *malware* é um *keylogger*, essa análise pode permitir que se localize o arquivo de log do *keylogger* no sistema, descobrir os tipos de registros que mantém, decifrar onde ele envia as suas informações, e assim por diante. Esse tipo de visão seria mais difícil de obter, utilizando apenas técnicas básicas estáticas. Nesta seção, descreve-se a técnica de análise dinâmica baseada em (SIKORSKI; HONIG, 2012).

2.3.8 *SANDBOXES*

Vários produtos de *software* são utilizados para executar análise dinâmica, principalmente por meio da tecnologia de *sandbox*. Uma *sandbox* é um mecanismo de segurança para executar programas não confiáveis em um ambiente seguro sem prejudicar sistemas "reais". *Sandboxes* compreendem ambientes virtualizados que muitas vezes simulam serviços de rede para garantir que o software ou *malware* que está sendo testado funcione normalmente. Muitas *sandbox* — tais como *Norman Sandbox*, *GFI Sandbox*, *Anubis*, *Joe Sandbox*, *ThreatExpert*, *BitBlaze*, *Cuckoo Sandbox* e *Comodo Instant Malware Analysis* — analisam *malware* gratuitamente (SIKORSKI; HONIG, 2012).

As *sandboxes* para análise de *malware* podem ser dividas em dois tipos: sandbox on-line e sandbox standalone (MANGIALARDO, 2015).

Um sandbox on-line é um ambiente de *sandbox* para análise de *malware* que disponibiliza o serviço na Internet. O usuário do serviço não pode alterar as configurações definidas pelo provedor. Este fornece normalmente uma interface para o usuário submeter o arquivo para análise. Ao fim da análise, gera-se o resultado com as informações e a situação do arquivo. São exemplos de *sandboxes on-line*:

- Malwr (https://malwr.com/);
- ThreatExpert (http://www.threatexpert.com/submit.aspx);
- Anubis (http://anubis.iseclab.org/);
- ThreatTrack ThreatAnalyser (http://www.threattracksecurity.com/resources /sandboxmalware- analysis.aspx);
- Comodo (http://camas.comodo.com/);
- Malbox (http://malbox.xjtu.edu.cn/);
- Xandora (http://www.xandora.net/xangui/).

Uma *sandbox standalone* é configurável e mantida por quem a implementou. Para isso, faz-se necessário configurar máquinas virtuais que criem o ambiente isolado e seguro. Além disso, utiliza-se um sistema como o *Cuckoo Sandbox* (GUARNIERI, 2016) para gerenciar as atividades de análise e geração de relatórios.

São exemplos de *sandboxes standalone*:

- Cuckoo Sandbox (http://www.cuckoosandbox.org/);
- Buster Sandbox (http://bsa.isoftware.nl/);
- Sandboxie (http://www.sandboxie.com/);
- Zero Wine (http://zerowine-tryout.sourceforge.net/);
- Remnux (http://zeltser.com/remnux/).

Entre esses, destaca-se o Cuckoo *Sandbox*, que possibilita processar qualquer arquivo suspeito e, em questão de segundos, fornecer resultados detalhados sobre as atividades do arquivo dentro de um ambiente isolado. Desse modo, permite-se entender seu comportamento, a fim de compreender o contexto, as motivações e os objetivos de uma violação, para melhor proteção no futuro. *Cuckoo Sandbox* é um *software* livre que automatiza a tarefa de analisar qualquer arquivo malicioso. De modo geral, ele é capaz de executar as seguintes funcionalidades (GUARNIERI, 2016):

- Analisar diferentes arquivos maliciosos (executáveis, *exploits* documentos, *applets Java*), bem como sites mal-intencionados, em ambiente *Windows, OS X, Linux e Android* virtualizados;
- Rastrear chamadas de API e comportamento geral de arquivos;
- *Dump* e análise de tráfego de rede, mesmo quando criptografadas;
- Realizar a análise avançada de memória do sistema virtualizado infectado com suporte integrado ao *Volatility*.

2.3.9 ANALISANDO PROGRAMAS *WINDOWS* MALICIOSOS

A maioria dos *malware* possui como alvo plataformas *Windows* e estabelece uma colaboração intensa com o sistema operacional. Uma compreensão da arquitetura do *Windows* permite identificar possíveis indicadores de *malware*. Desse maneira, monitorar a forma que os artefatos acessam o sistema operacional viabiliza determinar o propósito do *exemplar*.

Nesta seção, detalham-se algumas maneiras que os *malware* usam as funcionalidades do *Windows*. Essa análise concentrar-se nas funcionalidades mais relevantes para a análise de *malware*.

2.3.9.1 WINDOWS API

Windows API é um amplo conjunto de funcionalidades que permite que aplicações interajam com o sistema operacional. Esse conjunto é importante para análise de artefatos, uma vez que estes precisam acessar essas funções para executar suas atividades maliciosas.

Uma das formas mais comuns pela qual os *malware* interagem com o sistema operacional é criando e modificando arquivos. Atividades no sistema de arquivos podem revelar informações importantes acerca do comportamento do *malware*. Por exemplo, se o *artefato* cria um arquivo e armazena hábitos de navegação do usuário, o programa é provavelmente alguma forma de *spyware*.

A *Microsoft* provê várias funções para acessar o sistema de arquivo, como a *CreateFile* que é usada para criar ou abrir arquivos, ou a *WriteFile* empregada para escrever em arquivos.

2.3.9.2 REGISTRO DO WINDOWS

O registro do *Windows* é usado para armazenar informações do sistema operacional e configuração de programas. Assim como o sistema de arquivos, o registro é uma boa fonte de dados que pode revelar informações úteis sobre a funcionalidade do *malware*. As primeiras versões do *Windows* usava arquivos *ini* para armazenar informações de configuração. O registro foi criado como um banco de dados hierárquico de informações para melhorar o desempenho, e sua importância tem crescido à medida que mais aplicações utilizam para armazenar dados. Quase todas as informações de configuração do *Windows* são armazenadas no registro, incluindo redes, *driver*, *startup*, conta de usuário, entre outras. Frequentemente, *malware* usam o registro para dados de persistência ou de configuração. Os artefatos adicionam entradas no registro que permitem que ele seja executado automaticamente quando o computador inicia. De-

vido ao tamanho do registro, há várias formas de o *malware* usá-lo para persistência. Existem alguns termos importantes de registro:

- *Root key* - O registro é dividido em cinco seções de nível superior chamada de chaves-raiz. Às vezes, os termos *HKEY* e *hive* também são utilizados. Cada uma das chaves de raiz tem um propósito particular, conforme explicado a seguir.
- *Subkey* - É como uma subpasta dentro de uma pasta;
- *Key* - É uma pasta no registro que pode conter outras pastas ou valores. As *Root key* e *Subkey* são ambas *Key*;
- *Value entry* - É um par ordenado com um nome e valor;
- *Value ou data* - São os dados armazenados em uma entrada de registro.

O registro é dividido em cinco *Root key*:

- HKEY_LOCAL_MACHINE (HKLM) - Armazena configurações que são globais para a máquina local;
- HKEY_CURRENT_USER (HKCU) - Guarda as configurações específicas para o usuário atual;
- HKEY_CLASSES_ROOT - Mantêm informações de definição de tipos;
- HKEY_CURRENT_CONFIG - Armazena as configurações sobre a configuração de hardware atual, especificamente diferenças entre a configuração atual e a padrão;
- HKEY_USERS - Define configurações para o usuário padrão, novos usuários e o atual.

As duas chaves de raiz (*Root key*) mais frequentemente utilizadas são HKLM e HKCU. Algumas chaves são virtuais e fornecem uma maneira para fazer referência a informações de registro subjacente. Por exemplo, a chave é HKEY_CURRENT_USER realmente armazenada em HKEY_USERS\SID, em que SID é o identificador da segurança do usuário conectado no momento. Por exemplo, uma sub-chave popular, HKEY_LOCAL_

MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run, contém uma série de valores que são executados automaticamente quando um usuário se loga. A chave raiz é HKEY_LOCAL_MACHINE, que armazena as subchaves do software, Microsoft, Windows, CurrentVersion e Run.

Malware usa frequentemente funções de registro que são parte da API do *Windows* a fim de executar automaticamente quando o sistema é inicializado. Seguem-se as funções mais comuns do registro:

- *RegOpenKeyEx* - Abre um registro para edição e consulta. Existem funções que permitem consultar e editar uma chave de registro sem abri-lo, mas a maioria dos programas usam *RegOpenKeyEx* de qualquer maneira;
- *RegSetValueEx* - Adiciona um novo valor ao registro e define seus dados;
- *RegGetValue* - Retorna os dados para uma entrada de valor no Registro.

Ao observar essas funções em um malware, deve-se identificar o registro-chave que está sendo acessado. Além das chaves de registro para executar na inicialização, muitos valores de registro são importantes para a segurança e as configurações do sistema.

2.3.9.3 *NETWORKING APIs*

Malware, normalmente, baseia-se em funções de rede para fazer seu trabalho, e há muitas funções da API do *Windows* para comunicação de rede. É importante reconhecer e compreender as funções de rede comuns, para que se possa identificar o que um programa mal-intencionado está fazendo quando essas funções são acessadas.

Das opções de rede do *Windows*, *malware* frequentemente usa *Berkeley* soquetes, funcionalidade que é quase idêntica em sistemas *Windows* e *Unix*. A funcionalidade da rede *Berkeley* soquetes no *Windows* é implementada nas bibliotecas *Winsock*, principalmente em *ws2_32.dll*. Dessas, as funções *socket, connect, bind, listen, accept, send, e recv* são as mais comuns.

Há sempre dois lados para um programa de rede: o lado do servidor, que mantém um *socket* a espera de conexões de entrada, e o lado do cliente , que se conecta a esse

socket. *Malware* pode estar em um desses. No caso de aplicações do lado do cliente que se conectam a um *socket* remoto, existirá um chamada de *socket* seguido por uma chamada *connect*, seguido pelo *send* e *recv* quando necessário. Para um aplicação que "escuta"conexões de entrada, as funções socket, bind, listen e accept são chamadas nessa ordem, seguido pelo *send* e *recv*, conforme necessário. Esse padrão é comum tanto para software maliciosos quanto para não maliciosos.

Além do API *Winsock*, existe uma API de nível superior chamado de *WinINet API* API. As funções de API do *WinINet* são armazenados em *Wininet.dll*. Se um programa importa funções dessa DLL, ele está usando APIs de rede de nível superior. A API *WinINet* implementa protocolos como HTTP e FTP na camada de aplicação. Pode-se ganhar uma compreensão do comportamento do *malware* com base nas conexões que se abrem.

- *InternetOpen* é usado para inicializar uma conexão com a *Internet*;
- *InternetOpenUrl* é utilizado para conectar a um URL (que pode ser uma página HTTP ou um recurso FTP);
- *InternetReadFile* funciona semelhante à função *ReadFile*, permitindo que o programa possa ler dados de um arquivo baixado da Internet.

Malware pode usar a *API WinINet* para se conectar a um servidor remoto e obter mais instruções para a execução.

2.3.10 MONITORAMENTO DE CHAMADA DE FUNÇÃO

Conforme explicitado nas seções anteriores, monitorar as funções da API do *Windows* possibilita inferir características do comportamento de *software*. Nesta seção, baseada em (EGELE et al., 2012), é descrito o monitoramento das funções da API do *Windows* .

Uma função consiste de um código que executa uma tarefa específica, tal como, a criação de um arquivo. Embora a utilização de funções pode facilitar a reutilização de código e tornar o processo de manutenção mais simples, a propriedade que as faz

interessantes para a análise de programa é que elas são normalmente usadas para abstrair os detalhes de implementação. Quando se trata de análise de código, tais abstrações ajudam a ganhar uma visão geral do comportamento do programa. Assim, uma possibilidade para monitorar as funções chamadas por um programa é interceptar essas chamadas. O processo de interceptar chamadas de função é denominado de *hooking*. O programa analisado é manipulado de uma maneira que, ao chamar a função pretendida, a função de *hooking* é invocada. Essa função agora é responsável por implementar a funcionalidade requerida, como a gravação de um log, ou criação de um arquivo.

2.3.10.1 INTERFACE DE PROGRAMAÇÃO DE APLICAÇÃO

Funções que formam um conjunto coerente de funcionalidades, tais como manipulação de arquivos ou comunicação através da rede, são muitas vezes agrupadas em uma interface de programação de aplicação (ou API do Inglês *Application Programming Interface*). Os sistemas operacionais geralmente oferecem muitas APIs que podem ser usadas por aplicativos para executar tarefas comuns. Essas APIs estão disponíveis em diferentes camadas de abstração. Acesso de rede , por exemplo, pode ser fornecida por uma API que foca no conteúdo transmitido em pacotes TCP, ou por uma API de nível inferior que permite que o aplicativo crie e escreva pacotes diretamente em um *socket*. Em sistemas operacionais Windows, o termo Windows API refere-se a um conjunto de APIs que fornecem acesso a diferentes categorias funcionais , tais como redes, segurança, serviços de sistema e gerenciamento dos dispositivos.

2.3.10.2 CHAMADAS DE SISTEMA

Uma chamada de sistema (ou do inglês, *system call*) é o procedimento no qual um *software* solicita um serviço ao sistema operacional. Programas em sistemas computacionais normalmente são executados em dois modos: usuário ou *kernel*. Enquanto aplicações como processadores de texto ou programas de manipulação de imagem são executados no modo de usuário, o sistema operacional é executado em modo kernel. Somente o código que é executado em modo kernel tem acesso direto ao estado do sistema. Essa separação impede que os processos de modo de usuário interajam com o sistema e seu ambiente diretamente. Por exemplo, não é possível para um processo no modo usuário abrir ou criar um arquivo de modo direto. Em vez disso, o sistema

operacional fornece uma API especial — interface de chamada de sistema. Usando chamadas de sistema, um aplicativo de modo de usuário pode solicitar ao sistema operacional para executar um conjunto limitado de tarefas em seu nome. Assim, para criar um arquivo, um aplicativo de modo usuário precisa invocar a chamada de sistema específica indicando caminho, nome e método de acesso do arquivo. Uma vez que a chamada de sistema é invocada, o sistema é comutado para o modo de kernel (ou seja, código do sistema operacional é executado). Após a verificação de que a aplicação possui acesso suficiente para a ação desejada, o sistema operacional realiza a tarefa em nome do aplicativo de modo de usuário. No caso do exemplo de criação de arquivo, o resultado da chamada é um identificador de arquivo, onde todas as interações adicionais do aplicativo de modo de usuário em relação a este arquivo (por exemplo, gravar o arquivo) é realizada por meio desse identificador. Um *malware*, tal como qualquer outra aplicação, ao executar no espaço do usuário, precisa invocar as respectivas chamadas de sistema . Essas chamadas são a única possibilidade para um processo no modo usuário interagir com o *kernel* do sistema operacional. Logo essa interface é especialmente interessante para análise dinâmica de malware. No entanto, amostras de *malware* são conhecidas por conseguirem obter privilégios de modo *kernel*, sem fazer uso da interface de chamada de sistema em modo usuário.

2.3.10.3 WINDOWS NATIVE API

A *Windows Native API* é a interface real para o sistema Windows NT. Nesse sistema, a API Win32 é apenas uma camada acima da *Native API*. Em termos de funcionalidade, a *Native API* é a interface mais próxima do *kernel* do *Windows*, proporcionando interfaces de acesso direto ao gerenciador de memória e de I/O do sistema, bem como ao gerenciador de objetos, processos e threads, entre outros (EILAM, 2011).

O Core da *API Win32* contém cerca de 2.000 APIs, a depender da versão específica do Windows e se considera ou não APIs não documentadas. Essas APIs são divididas em três categorias: *Kernel*, *USER* e *GDI*. A Figura 2.1 mostra a relação entre as DLLs Win32, NTDLL.DLL e os componentes do kernel.

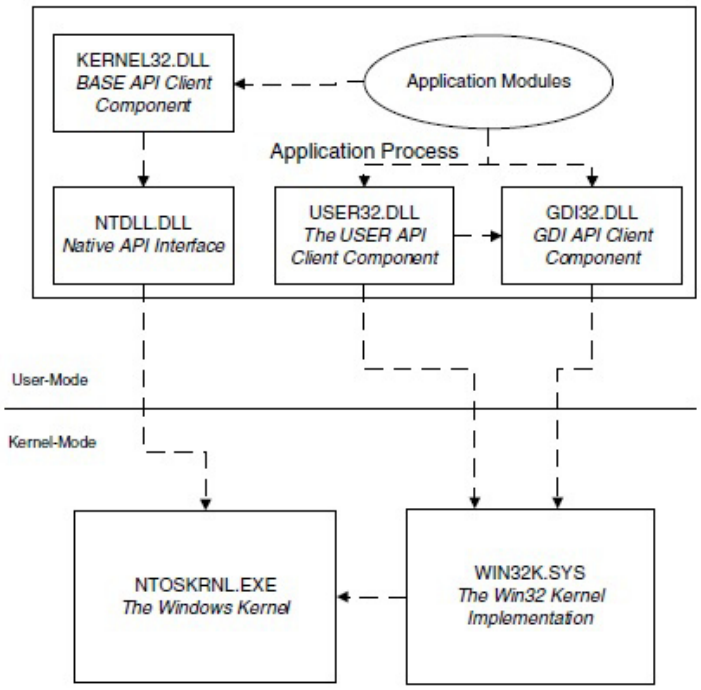

Figura 2.1: As DLLS Win 32 e suas relações com os componentes do kernel (EILAM, 2011)

Aplicações do usuário têm acesso a APIs do usuário, como *kernel32.dll* e outras DLLs, que chamam a *ntdll.dll*, uma DLL especial que gera as interações entre o espaço de usuário e de kernel. O processador então muda para o Modo de *kernel* e executa uma função no kernel, normalmente localizado em *Ntoskrnl.exe*. O processo é complexo, mas a separação entre o *kernel* e de APIs de usuário permite alterar o *kernel* sem afetar aplicações existentes.

As funções *NTDLL* usam *APIs* e estruturas em nível *kernel*. Essas funções compõem a *API* nativa. Os programas não devem fazer chamadas de API nativa, mas nada no SO impede de fazê-las.

Chamar a *API* nativa diretamente é atraente para os criadores de malware, porque lhes permitem fazer coisas que não seriam possíveis caso acessassem via *kernel32.dll*. Há muitas funcionalidades que não estão expostas na tradicional API do *Windows*, mas que podem ser atingidas chamando a API nativa diretamente.

Além disso, chamar a API nativa diretamente às vezes é mais furtivo. Muitos produtos antivírus e de proteção de acolhimento monitoram as chamadas de sistema feitas por um processo. Se o processo chama a função API nativa diretamente, pode ser capaz de escapar de um produto de segurança mal projetado.

Na Figura 2.2, apresenta-se um diagrama de uma chamada de sistema com um programa de segurança monitorando apenas as chamadas para o *Kernel32.dll*. A fim de contornar o programa de segurança , o *malware* usa a API nativa diretamente. Em vez de chamar as funções do Windows *ReadFile* e *WriteFile*, esse malware chama a função *NtReadFile* e *NtWriteFile*. Essas funções estão em *ntdll.dll* e não são monitoradas pelo programa de segurança. Um programa de segurança bem projetado monitora chamadas em todos os níveis, incluindo o *kernel*, para garantir que essa estratégia não funcione.

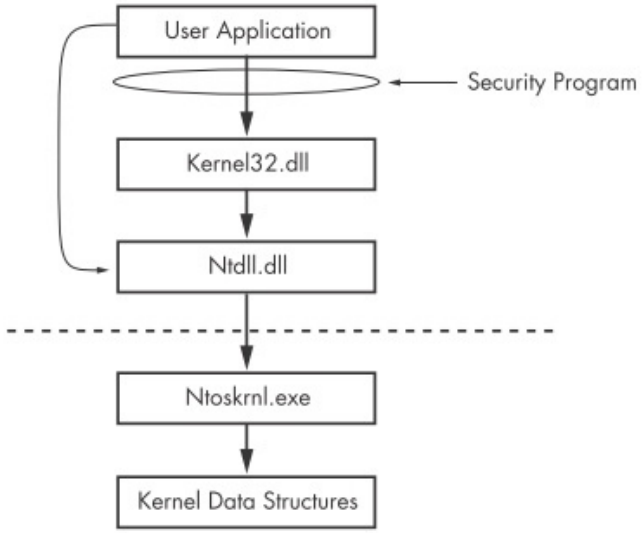

Figura 2.2: Usando a Native API para evitar detecção (SIKORSKI; HONIG, 2012)

Há várias chamadas da *native API* que podem ser usadas para obter informações sobre o sistema, processos, *threads, handles* e outros itens. Entre elas, vale destacar *NtQuerySystemInformation, NtQueryInformationProcess, NtQueryInformationThread, NtQueryInformationFile, e NtQueryInformationKey*. Estas chamadas fornecem informações mais detalhadas do que quaisquer chamadas da *API Win32*, e algumas funções permitem definir atributos refinados para arquivos, processos, *threads*, e assim por diante. Outra função da *native API* que é popular com os autores de malware é a *NtContinue*. Essa função é usada para retornar de uma exceção, destina-se a transferir a execução de volta para a *thread* principal de um programa depois de uma exceção tratada. No entanto, se o local de retonro for especificado no contexto da exceção, isso pode ser alterado. *Malware*, muitas vezes, usam essa função para transferir a execução de maneiras complexas, a fim de confundir um analista e fazer um programa mais difícil para analisar. As aplicações nativas são aquelas que não usam o subsistema *Win32* e exigem apenas a *native API*. O cabeçalho PE indica se um programa é um aplicativo

nativo.

2.3.10.4 *HOOKING* DE FUNÇÃO

Ao realizar *Hooking* em funções de API, consegue-se monitorar o comportamento de um programa no nível de abstração da respectiva função. Enquanto observações podem ser feitas pelo *hooking* de funções de API do Windows, uma vista mais detalhada do mesmo comportamento pode ser obtido por monitorização da *Native API*. O fato de que as aplicações de usuário precisam invocar chamadas de sistema para interagir com o sistema operacional, essas chamadas merecem atenção. No entanto, essa restrição só se aplica à execução de *malware* no modo usuário. Execução de *Malware* em modo *kernel* pode chamar, diretamente, as funções desejadas sem passar pela interface de chamada de sistema que se encontra em modo usuário.

2.3.11 ANÁLISE DE PARÂMETRO DE FUNÇÃO

Na análise estática, tenta-se inferir o conjunto de possíveis valores dos parâmetros das funções ou seus tipos de maneira estática. Já na análise dinâmica, foca-se no real valor que é passado quando a função é invocada. O rastreamento de parâmetros e valores de retorno de função permitem a correlação de chamadas de função individuais que operam no mesmo objeto. Por exemplo, se o valor de retorno (um identificador de arquivo) de um chamada de sistema *CreateFile* é utilizado em uma subsequente chamada de *WriteFile*, tal correlação é, obviamente, encontrada. Chamadas de funções agrupadas de forma lógica fornecem uma visão detalhada sobre o comportamento de um programa.

2.3.12 MONITORANDO O FLUXO DE INFORMAÇÃO

Uma abordagem ortogonal ao monitoramento de chamadas de função, durante a execução de um programa, é a análise de como o programa processa os dados. O objetivo de acompanhamento de fluxo de informação é observar a propagação de dados interessantes em todo o sistema, enquanto um programa que manipula esses dados é executado. Em geral, os dados que devem ser monitorados são, especificamente, marcados com uma etiqueta correspondente. Desse modo, sempre que os dados são processados

pela aplicação, seu rótulo é propagado. Além dos casos óbvios, políticas precisam ser implementadas para descrever como esses rótulos são propagados em cenários mais complexos.

2.3.13 MONITORANDO INSTRUÇÕES

Uma valiosa fonte de informação para um analista entender o comportamento de uma amostra é o monitoramento de instruções. Isto é, a sequência de instruções de máquina que a amostra executada enquanto está sendo analisada. Embora comumente seja difícil ler e interpretar esse tipo de informação, esse rastreamento pode conter informações importantes que não são representadas em um nível de abstração superior (por exemplo, chamadas de função).

2.3.14 PONTOS DE INÍCIO AUTOMÁTICO

Pontos de início automático (ASEPs (KING; CHEN, 2006)) definem mecanismos no sistema que permitem que os programas sejam, automaticamente, invocados no processo de inicialização do sistema operacional ou quando um aplicativo é iniciado. A maioria dos *malware* tenta persistir durante reinicializações de um hospedeiro infectado, adicionando-se a um dos disponíveis ASEPs. É, portanto, de interesse para o analista monitorar tais ASEPs quando uma amostra desconhecida é analisada.

Neste capítulo, realizou-se a fundamentação teórica da dissertação no que concerne à análise de *malware*. Inicialmente, explicitaram-se as possibilidades de classificação de códigos maliciosos. Em seguida, foram expostas as técnicas de análise de *malware*. Por fim, as análise estáticas e dinâmicas foram descritas. O capítulo seguinte apresenta um estudo dos métodos de aprendizado de máquina, mais especificamente dos algoritmos *K-Means* e *FCM* objetos desta dissertação.

3 APRENDIZADO DE MÁQUINA

3.1 INTRODUÇÃO

O aprendizado de máquina é um campo da inteligência artificial que permite processar e adquirir conhecimento de forma automática. No aprendizado de máquina, emprega-se a indução como forma de inferência lógica para obter conclusões genéricas sobre um con- junto de exemplos particular (MITCHELL, 1997).

O aprendizado indutivo é efetuado a partir de raciocínio sobre exemplos fornecidos por um processo externo ao sistema de aprendizado. O aprendizado indutivo pode ser dividido em supervisionado e não supervisionado. No primeiro, é fornecido ao algoritmo de aprendizado, ou indutor, um conjunto de exemplos de treinamento para os quais o rótulo da classe associada é conhecido. Em geral, cada exemplo é descrito por um vetor de valores de características, ou atributos, e o rótulo da classe associada. O objetivo do algoritmo de indução é construir um classificador que possa determinar corretamente a classe de novos exemplos ainda não rotulados, ou seja, exemplos que não tenham o rótulo da classe. Para rótulos de classe discretos, esse problema é conhecido como classificação e para valores contínuos como regressão. Já no aprendizado não supervisionado, o indutor analisa os exemplos fornecidos e tenta determinar se alguns deles podem ser agrupados de alguma maneira, formando agrupamentos ou *clusters*. Após a determinação dos agrupamentos, normalmente, é necessária uma análise para determinar o que cada agrupamento significa no contexto do problema que está sendo analisado (MONARD; BARANAUSKAS, 2003).

A aprendizagem não supervisionada é frequentemente mais desafiadora. A análise tende a ser mais subjetiva, e não há um simples objetivo para a análise, tal como a previsão de uma resposta. Aprendizagem não supervisionada é muitas vezes realizada como parte de uma análise exploratória de dados. Além disso, pode ser difícil avaliar os resultados obtidos a partir de métodos de aprendizagem não supervisionada, uma vez que não existe mecanismo universalmente aceito para validação de resultados sobre um conjunto de dados independente ((JAMES et al., 2013)).

Neste capítulo, serão abordados duas técnicas de aprendizagem não supervisionada, mais especificamente o processo de clusterização e análise de componentes principais, ambos baseados em (JAMES et al., 2013). Ademais, serão descritos a clusterização à luz dos conceitos da lógica *fuzzy*, bem como dois métodos de validação do número de clusters (cotovelo e silhueta).

3.2 ANÁLISE DE COMPONENTES PRINCIPAIS

Esta seção, baseada em (ZUBEN; ATTUX, 2010), aborda os fundamentos de análise de componentes principais.

A análise de dados pertencentes a espaços de alta dimensão é uma constante nas mais diversas aplicações de aprendizado de máquina. Em tais casos, pode ser complexo projetar um classificador ou um regressor com tantos parâmetros, ou pode mesmo ser difícil simplesmente realizar uma análise dos dados. Razões como essas fazem com que seja bastante relevante realizar processos de redução de dimensionalidade, o que pode também se vincular à ideia de compressão. Uma possibilidade clássica nesse sentido é obter uma transformação linear que possibilite a representação de dados N-dimensionais em um espaço M-dimensional de menor dimensão. Em outras palavras, a partir de uma série de vetores de dados $\mathbf{x_k} \in \mathbf{R^N}$, deseja-se obter uma transformação linear A que faça uma projeção desses dados, gerando vetores $\mathbf{z_k} \in \mathbf{R^M}$. Considere que os vetores $\mathbf{x_k}$ e $\mathbf{z_k}$ são vetores-coluna, de modo que A será uma matriz M x N. Nessa abordagem, há dois pontos importantes a considerar: 1) como conseguir uma projeção adequada, ou seja, representativa? 2) Quanta "informação" será perdida quando os dados forem levados para uma dimensão menor? Um modo natural para construir uma matriz de projeção é analisar qual seria o erro quadrático médio (EQM) amostral entre os dados projetados e os dados originais. Inicialmente, faz-se uma hipótese que não altera o grau de generalidade da análise: supor que os dados têm média amostral igual a zero. Se os dados tiverem média (note que a média, aqui, corresponde a um vetor) não nula, o caso acima é atingido se fizer, para todos os dados:

$$\mathbf{x_k} < -\mathbf{x_k} - \frac{\mathbf{1}}{\mathbf{Ndados}} \sum_{\mathbf{i=1}}^{\mathbf{Ndados}} \mathbf{x_i} \tag{3.1}$$

sendo N dados o número total de padrões. Perceba que a equação 3.1 significa simplesmente que é subtraída de cada dado a média amostral, forçando, assim, uma situação

de média zero. Nesse caso, a ideia exposta acima leva à seguinte função custo para que se obtenham as direções de projeção ótimas:

$$\mathbf{J_{PCA}} = \frac{\mathbf{1}}{\mathbf{Ndados}} \sum_{\mathbf{i=1}}^{\mathbf{Ndados}} \left\| \sum_{\mathbf{c=1}}^{\mathbf{M}} \mathbf{z_{ic}} \mathbf{a_c} - \mathbf{x_i} \right\|^{\mathbf{2}} \tag{3.2}$$

Pode-se mostrar que o conjunto de direções $\mathbf{a_c}$, c = 1, ..., M, que minimiza essa função custo corresponderá ao conjunto dos M autovetores associados aos M maiores autovalores da matriz de autocorrelação amostral $\mathbf{R_x}$ dos dados. Essa matriz, que é simétrica, tem dimensão N x N e é definida por elementos $\mathbf{r_{ij}}$ do tipo:

$$\mathbf{r_{ij}} = \frac{\mathbf{1}}{\mathbf{Ndados}} \sum_{\mathbf{l=1}}^{\mathbf{Ndados}} \mathbf{x_{l,i}} \mathbf{x_{l,j}} \tag{3.3}$$

sendo $\mathbf{x_{k,i}}$ o i-ésimo elemento do vetor $\mathbf{x_k}$. É possível definir essa matriz de modo ainda mais simples como:

$$\mathbf{R_x} = \frac{\mathbf{1}}{\mathbf{Ndados}} \sum_{\mathbf{l=1}}^{\mathbf{Ndados}} \mathbf{X_l} \mathbf{X_l^T} \tag{3.4}$$

Caso os dados não tivessem sido manipulados para terem média amostral nula, em vez da matriz de autocorrelação, trabalhar-se-ia com a matriz de autocovariância. No entanto, é sempre possível realizar a manipulação descrita em 3.1, de modo que a explicação apresentada permanece plenamente abrangente. Portanto, com base no problema formulado e analisado, obtém-se um receituário simples para as projeções ótimas:

1. Faça com que os dados passem a ter média amostral nula usando 3.1;
2. Estime a matriz de autocorrelação dos dados usando 3.4;
3. Escolha os M autovetores associados aos M maiores autovalores para fazerem o papel de direções de projeção. O valor de M é definido pelo usuário; e
4. Componha a matriz de projeção A concatenando os M vetores-coluna obtidos. Cada vetor corresponderá a uma direção de projeção.

Um primeiro ponto relevante é que os autovetores da matriz de autocorrelação são ortogonais, o que quer dizer que as direções de projeção possuem essa propriedade. Um segundo ponto é que a matriz $\mathbf{R_x}$ é definida não negativa, sendo, aliás, muitas vezes, definida positiva. Portanto, seus autovalores serão sempre não negativos e comumente positivos (HAYKIN, 1996). Os elementos que compõem cada vetor z (vide equação 3.2) são chamados de componentes principais do vetor x a ele associado. Por esse motivo, a metodologia exposta acima recebe o nome de análise de componentes principais (PCA, do inglês principal component analysis) (HYVARINEN; KARHUNEN, 2001). O conceito de PCA surge, diretamente, à luz da ideia de buscar projeções que minimizem o erro quadrático médio de "compressão" (redução de dimensionalidade). No entanto, há outra interpretação para as projeções obtidas.

O primeiro componente principal será definido pela projeção de um vetor de dados $\mathbf{x_k}$ segundo a direção dada por $\mathbf{a_1}$, que é o autovetor associado ao maior autovalor de $\mathbf{R_x}$. Como $\mathbf{z_{1,k} = a_1^T x_k}$, a variância desse componente principal será $\mathbf{E[z_1, k^2] = a_1^T R_x a_1}$ (no cálculo da variância, consideram-se que os vetores de dados têm média nula). É possível mostrar que esse valor de variância será o maior alcançável para toda e qualquer direção de projeção. Mais ainda, se os autovetores tiverem norma unitária, mostra-se que $\mathbf{E[z_1, k^2]}$ será exatamente igual ao maior autovalor, ou seja, $\boldsymbol{\lambda_1}$. Isso revela que o primeiro componente principal é definido pela projeção que leva à maior variância do sinal projetado, o que significa que, de certa forma, trata-se de uma projeção que "preserva ao máximo" o "conteúdo de energia" do sinal. Interessantemente, o segundo componente principal também é obtido por meio da direção que, com a restrição de ser ortogonal à primeira, leva à maior variância.

Essa direção corresponde à do autovetor associado ao segundo maior autovalor. Aliás, $\mathbf{E[z_2, k^2] = \lambda_2}$ para norma unitária. Essa ideia se estende até o caso limite em que o número de componentes principais é igual ao número de elementos dos vetores de dados, sendo a última direção incluída exatamente a direção do autovetor associado ao menor autovalor. No caso em que os autovetores possuem norma unitária, pode-se mostrar que

$$\mathbf{J_{PCA}} = \sum_{\mathbf{i=M+1}}^{\mathbf{N}} \lambda_\mathbf{i} \tag{3.5}$$

Em outras palavras, o nível do erro quadrático médio amostral associado ao modela-

mento por M componentes principais é igual à soma dos N-M autovalores "deixados de fora". Naturalmente, se M = N, não há redução de dimensionalidade, e, portanto, não há erro de reconstrução. Uma outra forma interessante de avaliar a qualidade da compressão é analisar a medida:

$$\alpha = \frac{\sum_{\mathbf{i=1}}^{\mathbf{M}} \lambda_{\mathbf{i}}}{\sum_{\mathbf{i=1}}^{\mathbf{N}} \lambda_{\mathbf{i}}} \tag{3.6}$$

O valor de α vai de 0 a 1, sendo tanto maior quanto, no sentido da variância, for mais bem-sucedida a projeção. É importante perceber de que forma é possível obter uma redução de dimensionalidade bem-sucedida. Para tanto, é preciso que os dados estejam, no sentido da variância, concentrados num subespaço de dimensão menor que N. Isso fará com que um número $\mathbf{M < N}$ de projeções do sinal seja suficiente para capturar os aspectos essenciais de sua estrutura, levando a um valor de $\mathbf{J_{PCA}}$ baixo. A Figura 3.1, mostrada a seguir, ilustra uma possibilidade de redução de N = 3 para um subespaço com M = 2. PCA dependa da matriz de autocorrelação dos dados, uma vez que é exatamente essa correlação entre elementos do vetor que pode gerar "direções preferenciais" para projeção.

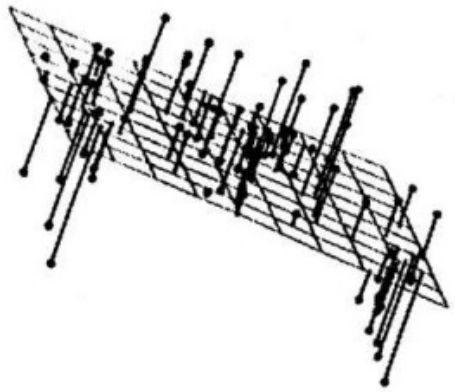

Figura 3.1: Exemplo de PCA com N = 3 e M = 2 (ZUBEN; ATTUX, 2010)

Após a redução de dimensionalidade, faz-se necessário decidir quantas componentes principais utilizar. Geralmente, não se está interessado em todos elas, em vez disso, usam-se apenas as primeiras componentes principais, a fim de visualizar ou interpretar os dados. Na verdade, o objetivo é utilizar o menor número de componentes principais que forneçam uma boa compreensão dos dados. Quantas componentes principais são

necessárias? Infelizmente, não há resposta simples a esta pergunta. Normalmente, escolhe-se o menor número de componentes principais que expliquem uma quantidade considerável dos dados. Isso é feito pela análise do gráfico, procura-se um ponto em que a contribuição da variância explicada pelas subsequentes componentes principais já não é tão significativa. No entanto, este tipo de análise visual é inerentemente subjetiva. Infelizmente, não há forma científica de decidir quantos componentes principais são suficientes. Na verdade, vai depender da área de aplicação e do conjunto de dados específico. Na prática, tem-se a tendência de olhar para os primeiros componentes principais, a fim de encontrar padrões interessantes nos dados. Se não há padrões interessantes nas primeiras principais componentes, torna-se improvável encontrar padrões relevantes nas posteriores. No entanto, se as primeiras componentes principais são interessantes, continua-se a observar as componentes principais seguintes, até não visualizar novos padrões significativos. Essa abordagem é reconhecidamente subjetiva, mas geralmente é usado como uma ferramenta para análise exploratória de dados.

3.3 CLUSTERIZAÇÃO

Clusterização, segundo (JAMES et al., 2013), refere-se a um amplo conjunto de técnicas para encontrar subgrupos, ou agrupamentos (*clusters*), em um conjunto de dados. Quando se clusteriza as observações de um conjunto de dados, procura-se dividi-las em grupos distintos de modo que as características dentro cada grupo sejam bastante semelhantes entre si, enquanto que as características em diferentes grupos sejam bastante dissemelhantes umas das outras. Naturalmente, é preciso definir o que significa que duas ou mais observações são semelhantes ou diferentes. Na verdade, trata-se, muitas vezes, de uma consideração específica de domínio que deve ser realizada com base no conhecimento dos dados a serem analisados. Por exemplo, suponha que se tem um conjunto de **n** observações, cada uma com **p** características. As **n** observações poderiam corresponder a amostras de tecido de pacientes com câncer de mama, e as **p** características poderiam ser medidas coletadas para cada amostra de tecido como o estágio do tumor ou seu grau, ou poderiam ser as medições de expressão de genes.

Pode haver alguma heterogeneidade entre as **n** amostras de tecido. Por exemplo, talvez existam diferentes subtipos de câncer da mama. *Clustering* poderia ser usado para encontrar esses subgrupos. Este é um problema não supervisionado, porque se tenta descobrir estruturas — nesse caso, *clusters* distintos — com base em um conjunto de dados, sem informações a respeito das saídas esperadas. No caso de problemas

supervisionados, já se conhece, *a priori*, as saídas esperadas para um determinado dado. *Clusterização* e *PCA* procuram simplificar os dados por meio de um número pequeno de resumos, mas os seus mecanismos são diferentes:

- PCA procura por uma representação de baixa dimensão das observações que explicam uma boa fração da variância;
- Clusterização tenta encontrar subgrupos homogêneos entre as observações.

Técnicas de agrupamento são populares em muitos campos de pesquisa. Desse modo, existe um grande número de métodos de agrupamento. Nesta seção, é descrito um dos métodos mais conhecidos de agrupamento: *K-Means*. Em seguida, o método *Fuzzy C-Means*, principal objeto deste trabalho, também é detalhado.

3.3.1 K-MEANS

A ideia de algoritmos não supervisionados é fornecer agrupamento de informações de acordo com os próprios dados, baseado em análises e comparações entre os seus valores numéricos. Dessa maneira, esses métodos agrupam os dados automaticamente, sem a necessidade de supervisão (BORGES, 2010).

Entre esses algoritmos, o K-Means (MACQUEEN, 1967) é amplamente utilizado. Tornou-se popular devido à sua simplicidade e tem sido, com sucesso, aplicado durante os últimos anos, sobretudo para agrupamento de *malware*. O objetivo desse método é particionar as amostras em K grupos, minimizando a distância *intra-clusters* e maximizando a distância *inter-clusters* (BORGES, 2010). A distância euclidiana constitui um critério de similaridade amplamente utilizado para as amostras no espaço euclidiano (ESTEVES; RONG, 2011).

Para executar *K-Means*, é preciso primeiro especificar o número desejado K de *clusters*. Então o *K-Means* vai atribuir cada observação a exatamente um destes. Os resultados vêm a partir de um simples e intuitivo problema matemático. Seja $\mathbf{C_1}, ..., \mathbf{C_k}$ os conjuntos contendo os índices das observações em cada *cluster*. Esses conjuntos deve satisfazer duas propriedades:

1. $\mathbf{C_1 \cup C_2 \cup ... C_K = \{1, ..., n\}}$ Cada observação deve pertencer a, pelo menos, um K *clusters*.

2. $\mathbf{C_K \cap C_{K'} = \varnothing}$ para todo $\mathbf{K \neq K'}$. Os clusters não se sobrepõem, ou seja, nenhuma observação pertence a mais de um cluster.

Por exemplo, se a i-ésima observação está no cluster de ordem k, então i $\in \mathbf{C_k}$. A ideia por trás do *K-Means* é que um bom agrupamento é aquele no qual dentro do *cluster* a variação é tão pequena quanto possível. A variação interna do *cluster* $\mathbf{C_k}$ é a medida $\mathbf{W(C_k)}$ do valor pelo qual as observações dentro de um cluster diferem uns das outras. Por isso, queremos resolver o problema

$$\mathbf{minimizar_{C_1,...,C_k}\{\sum_{k=1}^{k} W_{(C_k)}\}} \tag{3.7}$$

Da fórmula 3.7, pode-se inferir que se deseja particionar as observações em K *clusters* de modo que a variação dentro do conjunto total - a soma de todos K *clusters*- é tão pequena quanto possível.

É necessário definir variação interna no *cluster*. Há muitas possíveis formas de definir este conceito, mas a mais comum é a distância euclidiana quadrática:

$$\mathbf{W_{(C_k)} = \frac{1}{|C_k|} \sum_{i,i' \in C_k} \sum_{j=1}^{p} (x_{ij} - x_{i'j})^2} \tag{3.8}$$

Em que C_k denota o número de observações no cluster de ordem k. A variação dentro do *cluster* de ordem k é a soma das distâncias euclidianas entre as observações no *cluster* de ordem k, dividido pelo número total de observações nesse mesmo *cluster*. A combinação das expressões acima fornece o problema de optimização que define *K-Means*

$$\mathbf{minimizar_{C_1,...,C_k}\{\sum_{k=1}^{k} \frac{1}{|C_k|} \sum_{i,i' \in C_k} \sum_{j=1}^{p} (x_{ij} - x_{i'j})^2\}} \tag{3.9}$$

Diante dessa expressão, é preciso encontrar um algoritmo para resolvê-la, isto é, um método para particionar as observações em *clusters* k tal que a expressão é minimizada. Um algoritmo pode ser definido:

1. Aleatoriamente atribui um número de 1 a K, a cada uma das observações. Essas são atribuições iniciais de *cluster* para cada observação.
2. Executar as instruções abaixo até que as atribuições de *cluster* parem de mudar:
    - Para cada um dos k grupos, calcule o centroide. O centroide do *cluster* k_{th} é o vector de média de características para as observações no *cluster* k;
    - Atribuir cada observação ao *cluster* cujo centroide é o mais próximo, de acordo com a distância euclidiana.

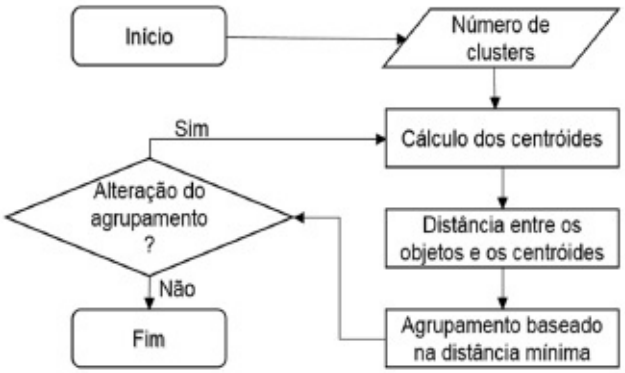

Figura 3.2: Fluxograma do K-Means (BAZZI; SOUZA; BETZEK, 2015)

3.3.2 LÓGICA FUZZY

Nesta seção é descrita a lógica *fuzzy* conforme abordada em (JANÉ, 2004).

A lógica *fuzzy* foi introduzida no contexto científico em 1965 pelo professor Lotfi Zadeh, por meio da publicação do artigo *Fuzzy Sets* (ZADEH, 1965) no *journal Information and Control*.

Com o transcorrer do tempo, a lógica fuzzy encontrou aplicação em várias áreas, por meio das quais tem mostrado sua capacidade de adaptação e facilidade de interface com o ser humano. Conforme (ALTROCK, 1996), pode-se encontrar aplicação para a

lógica *fuzzy* em diversas áreas como: avaliação de crédito, controle de fluxo de caixa, análise de risco, controle de estoques, avaliação de marketing, avaliação de fornecedores, controle de qualidade, otimização de inventários etc.

Ademais, outras aplicações tem feito uso dessa técnica segundo (YEN; LANGARI; ZADEH, 1995). São elas: controle automático de máquinas e equipamentos (controle de elevadores, tráfego automotivo, controle automático de foco em câmeras fotográficas, sistemas de acionamento robotizado etc.), otimização de processos produtivos, entre outros.

Considerado por (PINHO, 1999) como um novo ramo da matemática, a lógica fuzzy tem como ponto fundamental a representação da lógica humana na resolução de problemas complexos (ALTROCK, 1996). (CHIU; PARK, 1994) afirmam que, conforme o grau de incerteza de um problema aumenta, a capacidade de descrição de um modelo para resolução do mesmo decresce. Assim, fez-se necessário o surgimento de uma teoria que fornecesse subsídios para a resolução de problemas com alto grau de incerteza, sem que informações importantes se perdessem durante a manipulação dos dados por incapacidade do modelo matemático em lidar com a incerteza inerente às mesmas.

Nesse contexto, a lógica fuzzy é definida por (COX, 1995) como sendo capaz de combinar a imprecisão associada aos eventos naturais e o poder computacional das máquinas para produzir sistemas de resposta inteligentes, robustos e flexíveis. (KAUFMANN; GUPTA, 1988) afirmam que a lógica fuzzy é composta por conceitos e técnicas que dão a forma matemática ao processo intuitivo humano que na sua grande maioria é caracterizado pela imprecisão e ambiguidade. (ALTROCK, 1996) enfatiza que a lógica *fuzzy* permite o desenvolvimento de sistemas que representam decisões humanas, nas quais a lógica e a matemática convencional (*crisp*) se mostram insuficientes ou ineficientes. Desse modo, procura-se resolver problemas complexos e compostos por variáveis cuja informação contida é incerta, de uma maneira organizada e com a máxima confiabilidade possível.

A lógica fuzzy, por meio do uso de variáveis denominadas linguísticas (por exemplo, grande, pequeno, entre, ao redor de etc), consegue captar a incerteza associada a estas variáveis e traduzi-la para o modelamento matemático. Isto é possível, porque, diferentemente, da lógica *crisp*, a lógica fuzzy está baseada no conceito denominado grau de participação, ou função de pertinência (*membership*). Por exemplo, suponha que temos um conjunto de artefatos para serem distribuídos em 7 grupos possíveis :

vírus, *worm, backdoor, spyware, Bot, rootkit e trojan.* No caso da lógica *crisp*, cada artefato deve pertencer, obrigatoriamente, a apenas um grupo. Já na lógica *fuzzy*, um artefato pode pertencer a mais de um grupo, ou seja, um artefato pode estar no grupo de vírus e *trojan* ao mesmo tempo de acordo com a tabela de pertinência.

A estrutura de todo o sistema lógico fuzzy está baseada em três operações que estão explicitadas na Figura 3.3.

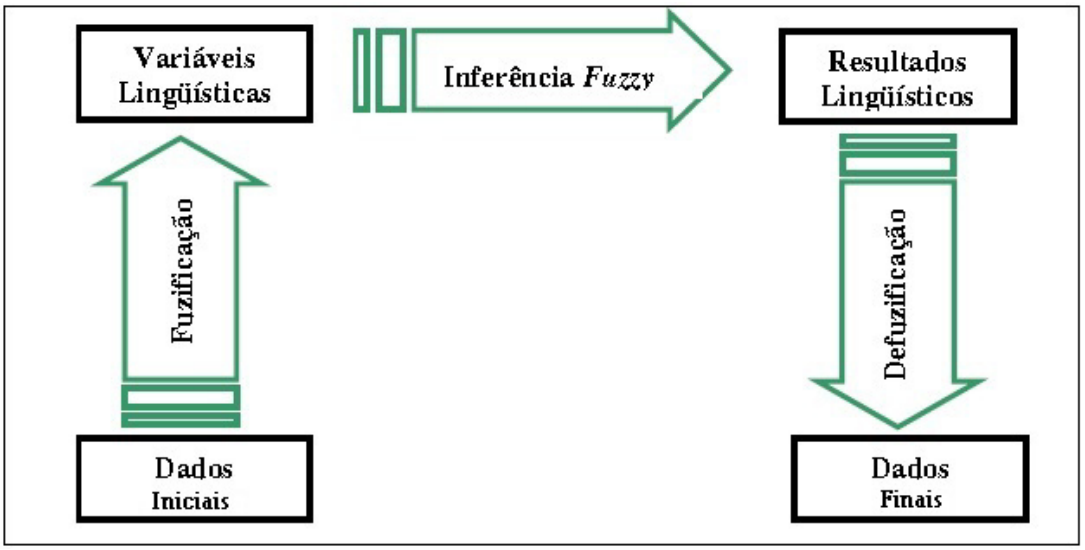

Figura 3.3: Sistema lógico *fuzzy* (COX, 1995)

3.3.2.1 FUZZIFICAÇÃO

O primeiro passo do sistema lógico *fuzzy*, chamado de fuzzificação, corresponde à transformação dos dados de entrada iniciais em suas respectivas variáveis linguísticas. Nessa etapa, todas as informações relativas à imprecisão ou incerteza associada a estas variáveis devem ser consideradas. (PINHO, 1999) cita a necessidade de que especialistas da área estudada sejam consultados durante a atribuição de valores relacionados aos graus de pertinência para cada uma das variáveis em estudo, contribuindo assim para maior precisão nos resultados.

3.3.2.2 INFERÊNCIA *FUZZY*

Uma vez feita a adequação dos valores iniciais em variáveis linguísticas, segue-se com a fase denominada inferência fuzzy, cuja finalidade é relacionar as possíveis variáveis entre si, por meio de regras pré-estabelecidas, cumprindo assim com os objetivos do algoritmo. Segundo (ALTROCK, 1996), pode-se separar esta fase em dois componentes visualizados na Figura 3.4 e denominados Agregação e Composição. O primeiro diz respeito à chamada parcela Se das regras que irão reger o processo de inferência, e o segundo refere-se à parcela Então do conjunto de regras assim chamadas, Se-Então. Tais componentes compõem o chamado processo de inferência lógica *fuzzy*, controlando as relações entre variáveis linguísticas por meio de seus respectivos operadores lógicos.

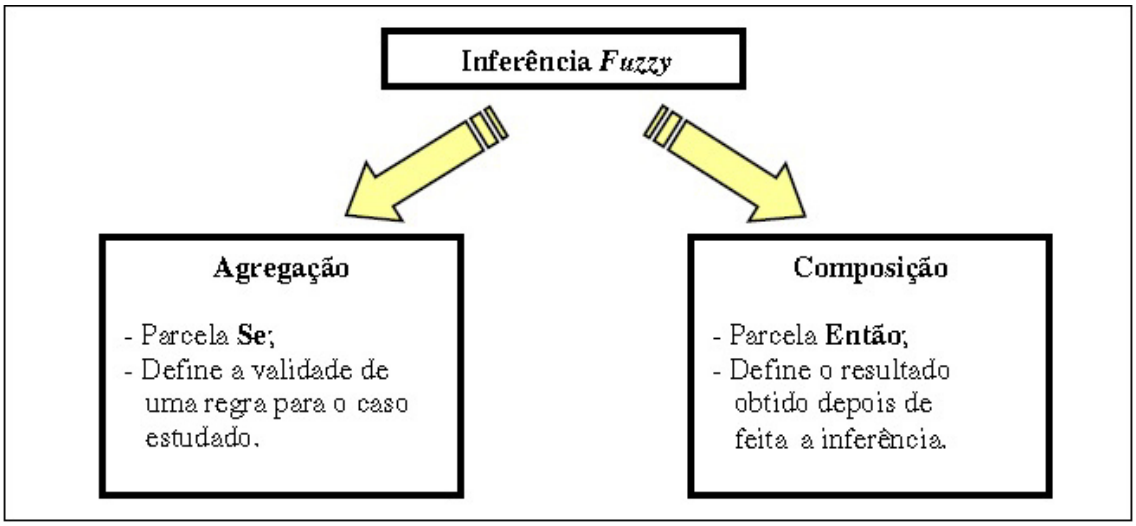

Figura 3.4: Inferência Fuzzy. (JANÉ, 2004)

3.3.2.3 DESFUZZIFICAÇÃO

A última fase do sistema lógico *fuzzy* é chamada desfuzzificação e compreende, segundo (ALTROCK, 1996), a tradução do resultado linguístico do processo de inferência fuzzy, em um valor numérico. Porém, (COX, 1995) fornece outra interpretação para o termo desfuzzificação, compreendendo o processo de conversão de um número fuzzy em um número real. Dentre os vários processos de desfuzzificação encontrados na literatura,

pode-se citar o método baseado em variáveis linguísticas e chamado de Centro-de-Máximo. Apresentado por (ALTROCK, 1996), esse método é caracterizado pela relação entre o valor linguístico e seu correspondente valor real. Em resumo, essa etapa consiste em retornar os valores, obter um valor numérico dentro da faixa estipulada pela lógica *fuzzy* (ZADEH, 1965).

3.3.3 FUZZY C-MEANS

Entre os algoritmos que utilizam lógica *fuzzy*, vale destacar o Fuzzy C-Means (FCM), objeto deste trabalho. Proposto por (DUNN, 1974) e estendido por (BEZDEK, 1981) é um método de agrupamento que permite estabelecer, para um certo dado, um determinado grau de relação com cada um dos agrupamentos obtidos. Ele possui funcionamento e estrutura semelhantes ao *K-Means*, mas possui uma abordagem *soft*, ao permitir que um dado não esteja associado exatamente com um único cluster. O grau de relação (também chamado de grau de pertinência) entre uma amostra e um agrupamento é um valor que está no intervalo [0, 1]. Uma pertinência próxima a 1 significa que o exemplar e o agrupamento em questão são similares. Caso contrário, se esse valor se aproxima de zero, indica dissimilaridade entre a amostra e o agrupamento analisado (BORGES, 2010).

Para a utilização do método de agrupamento, seguindo a mesma linha de (BAZZI; SOUZA; BETZEK, 2015), faz-se necessário determinar um expoente de imprecisão sendo que (ODEH; CHITTLEBOROUGH; MCBRATNEY, 1992) obtiveram resultados razoáveis com expoente entre 1,2 e 1,5, bem como (FRIDGEN et al., 2004) com expoente 1,5. Além do expoente de imprecisão, é necessário definir um critério de convergência (erro), que deve ser o mínimo possível. A classificação estará concluída quando o erro for menor que o configurado.

O objetivo da análise de agrupamento *Fuzzy* é minimizar estatisticamente a variabilidade dentro do grupo e, ao mesmo tempo, maximizar a variabilidade entre os grupos para gerar grupos homogêneos. Assim, uma amostra com vários atributos pode pertencer a diferentes grupos ao mesmo tempo, atribuindo adesão a diferentes grupos. A associação em cada classe é determinada por meio de um processo iterativo que começa com um conjunto aleatório de centróides de agrupamentos. Cada observação é atribuída para o mais próximo centróide e estes são reposicionados para cada grupo em função da média da distância das amostras do conjunto de dados. A distância Eucli-

diana é usualmente utilizada para calcular a distância das amostras de dados a fim de viabilizar o processo de agrupamento. (DAVATGAR; NEISHABOURI; SEPASKHAH, 2012).

O algoritmo *Fuzzy C-Means* considera um conjunto de dados $\mathbf{X} = \{\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_n}\}$ onde $\mathbf{x_k}$, corresponde a um vetor de características $\mathbf{x_k} = \{\mathbf{x_{k1}}, \mathbf{x_{k2}}, ..., \mathbf{x_{kp}}\} \in \mathbf{R^P}$ para todo K $\in \{\mathbf{1}, \mathbf{2}, ..., \mathbf{n}\}$ sendo $\mathbf{R^P}$ o espaço p-dimensional. Busca-se encontrar uma pseudo-partição *Fuzzy* que corresponde a uma família de c conjuntos *Fuzzy* de X , que representa a estrutura dos dados da melhor forma possível (BAZZI; SOUZA; BETZEK, 2015).

O algoritmo de classificação *Fuzzy C-Means* (Figura 3.5) orienta-se com parâmetros referentes ao número de agrupamentos que se deseja ter (c), uma medida de distância que define a distância permitida entre os pontos e os centróides $\mathbf{m} \in (\mathbf{1}, \infty)$ e um erro utilizado como critério de parada ($\varepsilon > \mathbf{0}$). A pertinência inicial é atribuída aleatoriamente, bem como os c centros iniciais que não devem possuir os mesmos valores iniciais devido a problemas que podem ocorrer durante a execução do algoritmo (FRIDGEN et al., 2004).

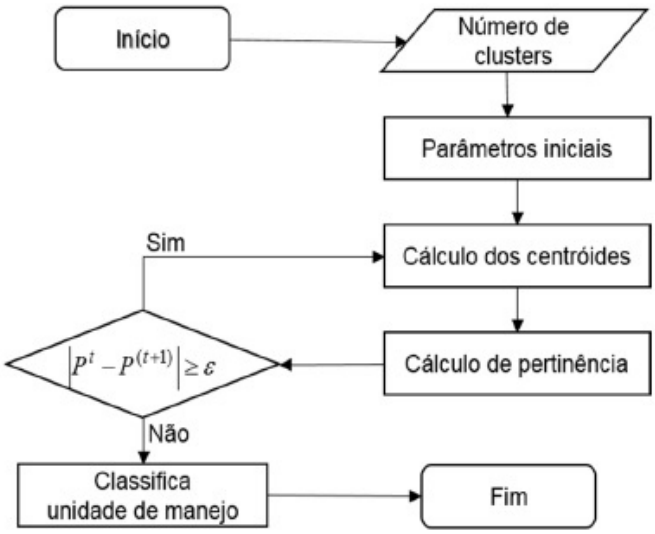

Figura 3.5: Fluxograma do algortimo Fuzzy C-Means. (BAZZI; SOUZA; BETZEK, 2015)

A posição de cada centroide é calculada considerando a distância passada por parâmetro inicialmente. Para cada c , calcula-se $\mathbf{v_1^t}, ..., \mathbf{v_c^t}$ pela expressão dada pela Figura 3.5) para a partição $\mathbf{P^t}$, sendo a iteração $\mathbf{t} = \{\mathbf{1}, \mathbf{2}, ..., \mathbf{n}\}$. Esse vetor corresponde ao centro do agrupamento $\mathbf{A_i}$ e é a média ponderada dos dados em $\mathbf{A_i}$. O peso do dado $\mathbf{x_k}$ é a

m-ésima potência do seu grau de pertinência ao conjunto Fuzzy $\mathbf{A_i}$.

$$\mathbf{v_i} = \frac{\sum_{\mathbf{k=1}}^{\mathbf{n}} [\mathbf{A_i}(\mathbf{x_k})]^{\mathbf{m}} \mathbf{x_k}}{\sum_{\mathbf{k=1}}^{\mathbf{n}} [\mathbf{A_i}(\mathbf{x_k})]^{\mathbf{m}}} \tag{3.10}$$

O cálculo do grau de pertinência do elemento $\mathbf{x_k}$ à classe $\mathbf{A_i}$ dada pela expressão na Equação 3.11 é realizado para cada $\mathbf{x_k} \in \mathbf{X}$ e para todo $\mathbf{i} \in \{\mathbf{1}, \mathbf{2}, ..., \mathbf{n}\}$ se $||\mathbf{x_k} - \mathbf{v_i^t}||^{\mathbf{2}} > \mathbf{0}$

$$\mathbf{A_i^{(t+1)}x_k} = \left[\sum_{\mathbf{j=1}}^{\mathbf{c}} \left(\frac{\|\mathbf{x_k} - \mathbf{v_i^t}\|^{\mathbf{2}}}{\|\mathbf{x_k} - \mathbf{v_j^t}\|^{\mathbf{2}}} \right)^{\frac{\mathbf{1}}{\mathbf{m-1}}} \right]^{\frac{\mathbf{-1}}{}} \tag{3.11}$$

em que, $||\mathbf{x_k} - \mathbf{v_i^t}||^{\mathbf{2}}$ representa a distância entre $\mathbf{x_k}$ e $\mathbf{v_i}$

Como critério de parada, compara-se $\mathbf{P^t}$ e $\mathbf{P^{(t+1)}}$. Se $|\mathbf{P^t} - \mathbf{P^{(t+1)}}| < \varepsilon$, o algoritmo é finalizado e a classificação é realizada considerando a pertinência gerada na última iteração. A Equação 3.11 retorna um vetor de valores de adesão para cada indivíduo, que indica o quão forte ele representa um típico membro de cada classe.

3.4 VALIDAÇÃO DE CLUSTERS

O processo de validação avalia os resultados de um agrupamento de forma objetiva e quantitativa (NASSIF, 2012). Essa tarefa é realizada para determinar o grau de significância dos resultados obtidos pelo algoritmo de agrupamento. O fato de não haver uma classe definida para os dados em tarefas de aprendizado não supervisionados acarreta na utilização de heurísticas e suposições para a identificação das estruturas embutidas nesses dados, que são avaliadas e validadas considerando-se a qualidade dos *clusters* encontrados pelos diversos algoritmos, ou seja, por meio da análise de índices que medem a adequabilidade da estrutura encontrada em termos das probabilidades dos *clusters* serem corretos. A adequabilidade dos *clusters* extraídos refere-se ao fato de representarem corretamente a informação ou a habilidade de recuperarem padrões que reflitam os conceitos intrínsecos ao conjunto de dados (METZ, 2006).

A validação do agrupamento, em geral, é realizada com base em índices estatísticos que julgam, de uma maneira quantitativa, as estruturas encontradas no conjunto de

dados. A maneira pela qual um índice é aplicado para validar um agrupamento é dada pelo critério de validação. Assim, um critério de validação expressa a estratégia utilizada para validar uma estrutura de agrupamento, enquanto que um índice é um valor estatístico pelo qual a validade é testada. Existem três tipos de critérios para investigar a validade de um agrupamento (HALKIDI; BATISTAKIS; VAZIRGIANNIS, 2002): relativos, internos e externos.

Os critérios relativos têm como objetivo encontrar o melhor agrupamento que um algoritmo pode obter sob certas suposições e valores para seus parâmetros, ou o algoritmo mais apropriado para os dados e estruturas analisadas. A maneira mais comum de aplicação de um índice com um critério relativo, por exemplo, consiste do cálculo do seu valor para vários agrupamentos que estão sendo comparados, obtendo-se uma sequência de valores. O melhor agrupamento é determinado pelo valor que se destaca nessa sequência, como o valor máximo, mínimo ou inflexão na curva do gráfico construído com a sequência.

Os critérios internos e externos são baseados em testes estatísticos e têm um alto custo computacional. Seu objetivo é medir o quanto o resultado obtido confirma uma hipótese pré-especificada. Nesse caso, são utilizados testes de hipótese para determinar se uma estrutura obtida é apropriada para os dados. O mesmo índice pode ser utilizado em um critério interno ou externo, embora as distribuições de referência do índice sejam diferentes.

Os índices externos medem o quão bem um agrupamento obtido se adequa a um agrupamento de referência, já conhecido anteriormente. Já os índices internos medem a qualidade de um agrupamento utilizando características internas dos próprios dados, sem empregar informações externas. (JAIN; DUBES, 1988). O que distingue a utilização de um índice em cada um dos critérios é a maneira como o índice é aplicado.

3.4.1 INTERPRETAÇÃO DE CLUSTERS

Os *clusters* encontrados pelo algoritmo são interpretados para tentar descobrir significados relacionados ao domínio da aplicação. A atribuição de conceitos aos agrupamentos é parte fundamental do processo de descoberta de conhecimento. Porém, é uma tarefa complexa que exige a participação do especialista do domínio na análise dos agrupamentos para que ele identifique os conceitos que, eventualmente, cobrem ou explicam

os exemplos contidos em um mesmo *cluster*. Assim, o especialista pode avaliar, subjetivamente, os *clusters* encontrados para descobrir a existência de algum significado prático, e/ou diferenças, considerando os padrões representados em cada *cluster*.

Pode ser difícil interpretar o resultado encontrado pelos algoritmos, pois não apresentam descrições conceituais simples, mas apenas um conjunto de agrupamentos dos dados, descritos frequentemente por meio de valores estatísticos e índices de similaridade. Essa limitação sugere que novas técnicas e ferramentas sejam utilizadas para tentar auxiliar o especialista no entendimento dos conceitos descritos pelos *clusters*. Para que o conhecimento extraído seja útil à aplicação, os padrões devem ser de fácil interpretação. Em (MARTINS, 2003) é proposta uma metodologia que faz uso de técnicas de aprendizado supervisionado para guiar a tarefa de explicação de *clusters*. A ideia básica é utilizar algoritmos de aprendizado supervisionado com o objetivo de descobrir e interpretar, com auxílio do especialista, os padrões encontrados pelos algoritmos de agrupamento sobre os dados não rotulados. A metodologia proposta para auxiliar a tarefa de interpretação de *clusters* de maneira semiautomática, ilustrada na Figura 3.6, consiste em uma sequência de estágios que compreende tanto algoritmos de aprendizado não supervisionado quanto algoritmos de aprendizado supervisionado (METZ, 2006). Essa metodologia é composta das seguintes etapas:

1. **Clustering** um conjunto de dados não rotulados é submetido a um algoritmo de agrupamento. O algoritmo utilizado é responsável por descobrir os clusters presentes nesse conjunto de dados.

2. **Rotulamento do conjunto de exemplos** o resultado obtido pelo algoritmo na etapa um é processado para que os dados do conjunto original, ou um subconjunto desses dados, sejam rotulados com um nome que identifica o *cluster* ao qual foram atribuídos durante o processo de *clustering*. Com isso, é criado um novo conjunto de dados com uma dimensão adicional, no qual o *cluster* é utilizado como o atributo classe.

3. **Indução** o conjunto de dados gerado na etapa dois possui as características necessárias para serem utilizados nessa etapa como entrada para algoritmos de aprendizado supervisionado, e, assim, obter uma descrição simbólica dos *clusters* identificados pelo algoritmo de agrupamento. Como o interesse é tentar explicar para o usuário/especialista os **clusters** previamente encontrados, a linguagem de descrição de conceitos utilizada pelo algoritmo de aprendizado supervisionado

deve ser uma linguagem simbólica de fácil entendimento, tal como regras ou árvores de decisão.

4. **Interpretação** o conhecimento do especialista do domínio é de fundamental importância para a interpretação conceitual dos *clusters*, agora descritos utilizando um formalismo simbólico e portanto de mais fácil entendimento. Com a interpretação do especialista, é possível ter uma explicação para os dados pertencentes a cada cluster encontrado pelo algoritmo de aprendizado não supervisionado.

Caso os resultados não sejam satisfatórios, pode-se decidir repetir o processo utilizando configurações diferentes dos agrupamentos encontrados, alterando o algoritmo de agrupamento.

Figura 3.6: Metodologia para interpretação de *clusters* (MARTINS, 2003)

É importante observar que um algoritmo de agrupamento sempre encontrará *clusters* em um conjunto de dados, independentemente dos conceitos por eles representados. No entanto, os *clusters* nem sempre descrevem um agrupamento adequado ou estão relacionados com os objetivos da aplicação. Ainda, os *clusters* não representam, necessariamente, uma descrição conceitual relacionada a uma "classe". Dessa maneira, dois ou mais *clusters* podem agrupar exemplos que se referem a um mesmo conceito (METZ, 2006).

3.4.2 DETERMINAR O NÚMERO DE CLUSTERS

Determinação do número de *clusters* em um conjunto de dados é um problema frequente em agrupamento de dados, sendo um problema distinto do processo de resolver o problema de agrupamento. Para uma determinada classe de algoritmos de agrupamento, em particular *K-Means* ou *FCM*, existe um parâmetro que especifica o número de agrupamentos (k). Outros algoritmos, como o agrupamento hierárquico, não requer que esse parâmetro seja informado antecipadamente (BATSOS et al., 2012).

A escolha correta de k é difícil, com interpretações dependendo da forma e da escala da distribuição de pontos em um conjunto de dados. Além disso, aumentar sempre reduz a quantidade de erro no agrupamento resultante, para o caso extremo de erro zero, se cada ponto de dados é considerado o seu próprio conjunto (isto é, quando k é igual ao número de pontos de dados, n). Intuitivamente, a escolha ideal de k vai encontrar um equilíbrio entre todos os dados em um único *cluster*, e cada ponto de dados no seu próprio *cluster*. Se não é possível inferir um valor apropriado de k a partir do conhecimento prévio das propriedades do conjunto de dados, ele deve ser escolhido utilizando algum método (OMATU et al., 2014). Existem métodos para tomar essa decisão. Entre esses métodos, dois serão detalhados por serem utilizados neste trabalho: método cotovelo e silhueta.

O mais antigo método para determinar o número de *clusters* em um conjunto de dados é chamado de método cotovelo. O método cotovelo verifica a percentagem de variância explicada como uma função do número de agrupamentos, isto é, deve-se escolher um certo número de agrupamentos de modo que a adição de outro agrupamento não forneça um modelo de dados muito melhor. Mais especificamente, traça-se a percentagem de variância explicada pelos grupos contra o número de *clusters*, os primeiros grupos vão acrescentar muita informação, mas em algum momento o ganho marginal vai cair, dando um ângulo no gráfico. O número de grupos é escolhida, neste ponto, pelo o "critério do cotovelo". Este "cotovelo"pode nem sempre ser identificados sem ambiguidade (KETCHEN; SHOOK, 1996). Nesse caso, percentagem de variância explicada é a relação entre a variação entre grupos pela variância total (GOUTTE et al., 1999).

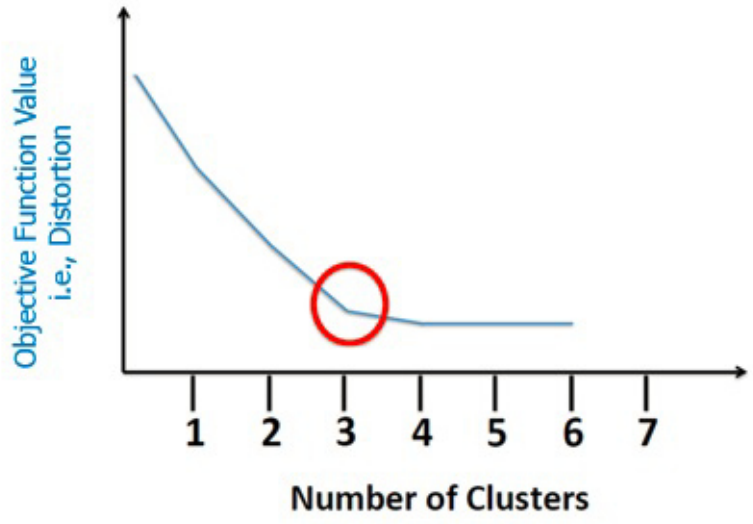

Figura 3.7: Variância explicada (EDUREKA, 2016)

O número de *clusters* é indicado pelo cotovelo na Figura 3.7. O número de *clusters* escolhidos deve ser 3.

O silhueta refere-se a um método de interpretação e validação de consistência dentro de grupos de dados. A técnica fornece uma representação gráfica sucinta de como cada objeto se encontra dentro de seu *cluster*. Foi descrita pela primeira vez por (ROUSSEEUW, 1987). O valor silhueta é uma medida de quanto um objeto é semelhante ao seu próprio agrupamento (coesão) em comparação com outros *clusters* (separação). Assume valores entre -1 a 1, em que um valor alto indica que o objeto está bem adaptado ao seu próprio *cluster* e mal correspondido aos *clusters* vizinhos. Se a maioria dos objetos têm um valor alto, então a configuração de *cluster* é apropriado. Se muitos pontos têm um valor baixo ou negativo, então o número de *cluster* pode ser superior ou inferior. A silhueta pode ser calculada com qualquer métrica de distância, tais como a distância euclidiana ou a distância de *Manhattan* (ROUSSEEUW, 1987).

3.4.3 FRAMEWORKS DE APRENDIZADO DE MÁQUINA

3.4.3.1 LINGUAGEM R

R é uma linguagem e ambiente para computação estatística e gráficos. É um projeto GNU, que é semelhante à linguagem S [9]. R fornece uma ampla variedade de técnicas estatísticas gráficas (modelagem linear e não linear, testes estatísticos clássicos, análise de séries temporais, classificação, agrupamento, entre outras) e é altamente extensível. Um dos pontos fortes da R é a facilidade com que gráficos podem ser produzidas, incluindo símbolos e fórmulas matemáticas, quando necessáris. R está disponível como software livre sob os termos da GNU *(General Public License of the Free Software Foundation)* em forma de código fonte. Ele compila e roda em uma ampla variedade de plataformas: *UNIX* e sistemas similares (incluindo *FreeBSD* e *Linux*), *Windows* e *MacOS* (GENTLEMAN; IHAKA, 2014).

R é um conjunto integrado de instalações de software para manipulação de dados, cálculo e visualização gráfica. Inclui as seguintes funcionalidades (GENTLEMAN; IHAKA, 2014):

[9] S é uma linguagem de programação estatística desenvolvida principalmente por *John Chambers*

- uma manipulação de dados eficaz;
- um conjunto de operadores para cálculos em matrizes;
- uma grande coleção integrada de ferramentas para análise de dados;
- instalações gráficas para análise de dados e visualização, quer na tela ou em cópia impressa;
- linguagem de programação simples e eficaz que inclui condicionais, loops, funções recursivas definidas pelo usuário e recursos de entrada e saída;
- biblioteca "e1071"específica para trabalhar com o método FCM.

Por fim, devido a todas essas características supracitadas, esta linguagem foi a escolhida para implementar o método deste trabalho.

3.4.3.2 WEKA

Weka é um conjunto de algoritmos de aprendizado de máquina para tarefas de mineração de dados. Os algoritmos podem ser aplicados, diretamente, a um conjunto de dados ou chamado a partir de seu próprio código Java. *Weka* contém ferramentas para pré-processamento de dados, classificação, regressão, *clustering*, regras de associação e visualização. É também bem adequada para o desenvolvimento de novos sistemas de aprendizagem de máquina. Trata-se de um software de código aberto sob a *GNU General Public License* (GROUP, 2016)

Possui uma série de heurísticas para mineração de dados relacionadas à classificação, regressão, clusterização, regras de associação e visualização, entre elas: *Naive Bayes, Linear Regression, IB1, Bagging, LogistBoot, Part, Ridor, ID3 e LMT*. Por ter sido desenvolvido usando a abordagem de *framework*, WEKA é extensível, permitindo que novos algoritmos ou funcionalidades sejam adicionadas de maneira relativamente fácil. Vale ressaltar que, para que uma base de dados seja carregada no software, é necessário que arquivo esteja no formato .arff (*Attribute Relation File Format*), formato de leitura para o *Weka* (BORGES, 2010).

O *Weka* possui interface gráfica. Cada pacote principal como Filtros, Classificadores, *Clusters*, associações e seleção de atributo é representado na interface gráfica junta-

mente com uma ferramenta de visualização que permite que conjuntos de dados e previsões de Classificadores e *Clusters* sejam visualizados em duas dimensões (GROUP, 2016).

4 MÉTODO PROPOSTO

Nos capítulos anteriores, discutiu-se a fundamentação teórica da dissertação, na qual foram abordados a análise comportamental de *malware* e o aprendizado de máquina necessários a propositura do modelo. Este capítulo destina-se à apresentação do método proposto.

4.1 ANÁLISE COMPORTAMENTAL

Nesta seção, discute-se, brevemente, as fases necessárias para a geração do vetor de características a ser usado na etapa de agrupamento com os algoritmos de aprendizado de máquina, bem como o ambiente de execução utilizado. Basicamente, todo o processo para se completar a análise comportamental é composto por três fases: coleta de exemplares de *malware*; captura das chamadas de funções das APIs; e geração do dicionário de termos.

No entanto, antes de expor a análise comportamental, faz-se necessário esclarecer o ambiente de execução, bem como uma breve descrição do programa de análise usado ao longo do projeto.

4.1.1 AMBIENTE DE EXECUÇÃO

A abordagem dinâmica de análise de *malware* necessita que o sistema tenha múltiplos ambientes confinados seguros onde as amostras possam ser processadas, possibilitando capturar seus comportamentos. Foram utilizadas máquinas virtuais (VM), uma que elas fornecem rápida recuperação de estado, APIs para controlar, automaticamente, os sistemas *guests*. Desse modo, as amostras foram processadas no sistema de análise dinâmica *Cuckoo Sandbox* executando no sistema operacional *Linux Ubuntu*. No *Cuckoo*, configuraram-se 3 máquinas virtuais com *Windows XP* atualizados com *Service Pack* 3. Além disso, para se aproximar de um ambiente real de usuário, *software*

comumente usados como *Acrobat Reader, Skype, Flash, Office*, entre outros foram instalados nas máquinas virtuais. Esses programas foram baixados e instalados por meio do *Ninite* (SWIESKOWSKI; KUZINS, 2016) Com relação ao hardware, utilizou-se a seguinte configuração: Intel Core i5-4300 2.50 GHz e Memória 8,0 GB.

4.1.2 CUCKOO SANDBOX

Cuckoo Sandbox foi o programa de análise de código-fonte aberto escolhido neste projeto, uma vez que proporciona interface de controle para várias VMs e a capacidade de personalizar seu. Com essa ferramenta é possível injetar *software* maliciosos para um Ambiente Virtual limpo e recolher suas informações comportamentais, salvando-o para uso posterior. Após a análise , é possível restaurar a máquina infectado para um estado limpo anterior, a fim de que ela possa ser utilizada para a infecção com uma outra nova amostra. Vale ressaltar que o tempo de execução de cada *malware* é limitado a, no máximo, 600 segundos. Após esse período, o artefato é finalizado e os processos de análise são terminados. Existem três principais razões pelas quais o processo de análise pode terminar antes do previsto (PIRSCOVEANU, 2015):

1. A amostra detecta que ele está sendo monitorado e termina para evitar análise;
2. O *malware* injetado pode ter um comportamento baseado em gatilho que denota apenas quando determinadas condições acontecem . Alguns artefatos podem executar apenas em determinados datas ou horas, ou mesmo aguardar que a conexão à Internet esteja disponível;
3. O código malicioso não é compatível com o sistema operacional instalado no *guest*.

Para os itens um e dois supracitados, supõe-se, neste projeto, que esse tipo de *malware* não executará mais de 50 chamadas de API. O item três representa uma análise de falha, onde a amostra não consegue executar na máquina de destino devido a problemas de incompatibilidade. Essa análise de falha é sempre marcada pelo *Cuckoo* com uma mensagem de erro indicando que a análise está incompleta Deve notar-se que existe uma diferença entre as duas situações. Nos dois primeiros, irão executar e criar apenas um único processo denotada pelo executável e, ao mesmo tempo, possuir uma pequena quantidade de chamadas de API; o último item não criará processo e nenhuma

informações comportamental estará disponível. Portanto, no *script* de leitura dos relatórios *JSON* gerados pelo *Cuckoo* (Apêndice A), excluíram-se amostras com menos de 50 chamadas. Esse número foi escolhido pelo fato de se ser difícil agrupar amostras com número de chamadas inferior a 50.

O gerenciador de banco de dados utilizado foi o *PostgreSQL* [10]. Inicialmente, usou-se o *SQLite*, no entanto, este apresentou instabilidade, quando do processamento do elevado volume de amostras.

4.1.3 COLETA DE *MALWARE*

Nesta fase, foram obtidas amostras de *malware* do *site VirusShare* [11]. Nesse site, é possível a busca de exemplares para *download* por ano de recebimento. Assim, foram separados, de forma aleatória, exemplares de 2012, 2013, 2014, 2015 e 2016, totalizando a quantidade de 21.455 amostras consideradas maliciosas.

4.1.4 CAPTURA DAS FUNÇÕES DE API

Os 21.455 exemplares de *malware* coletados foram submetidos para execução no sistema de análise dinâmica *Cuckoo Sandbox*. Durante cada execução, esse analisador monitorou as funções de API chamadas pelo artefato sob análise e gerou um relatório em formato *JSON* para cada amostra. Os relatórios contêm informações do *malware* obtidas de maneira estática e dinâmica pelo *Cuckoo*.

4.1.5 GERAÇÃO DO DICIONÁRIO DE TERMOS

Os relatórios, gerados anteriormente, passaram por um processo de filtragem (implementado por meio de um *script* em Python), usado para identificar as funções de API mais relevantes. Assim, para cada arquivo *JSON* gerado, levantou-se quais chamadas a funções de API são consideradas maliciosas. Para isso, as funções de API presentes nos relatórios gerados foram comparadas com 153 funções de API comumente utilizadas

[10] https://www.postgresql.org/
[11] https://virusshare.com/

por *malware* de acordo com o trabalho de (PIRSCOVEANU, 2015). Apenas as funções acessadas pelo exemplar que pertencem a essa lista de 153 chamadas de funções foram utilizadas para a geração do dicionário de termos. Assim, após a filtragem e comparação de todos os 21.455 relatórios produzidos a partir da execução dos exemplares do conjunto de dados, obteve-se um dicionário composto por 144 termos (ou chamadas de funções).

O *script* de filtragem, após criar o dicionário de termos, gera um arquivo em formato CSV (*comma-separated values*). Esse arquivo contém a quantidade de vezes em que uma dada função de API presente no dicionário de termos é chamada por cada um dos exemplares, conforme ilustrado na Tabela 4.1.

Tabela 4.1: Exemplo de quantidade de chamadas feitas a cada função contida no dicionário de termos (colunas) por exemplar do conjunto de amostras (linhas).

NtQueryValueKey	**LdrGetProcedureAddress**	**GetSystemMetrics**	**recvfrom**
25	430	4	0
22	508	147	0
1	514	36	0
3	474	33	0
7	299	0	0
27	580	21	0
1	484	6	0
1	607	4	0
30	1023	4	0
1	1091	184	0
1	400	4	0
12	465	8	2
73	867	49	0

4.2 APRENDIZADO DE MÁQUINA

Nesta seção, apresenta-se a aplicação dos algoritmos de aprendizado de máquina nos dados obtidos na etapa de análise comportamental. O arquivo CSV gerado com os vetores de características extraídos a partir de todos os exemplares executados serviu de entrada tanto para o algoritmo *K-Means* quanto para o *FCM*. Ambos foram implementados em linguagem "R". Porém, devido à grande dimensão do vetor de características

(144 termos) construído na etapa anterior, antes da aplicação dos algoritmos de agrupamento, é necessário um procedimento para redução de dimensionalidade, por meio da análise de componentes principais.

4.2.1 ANÁLISE DE COMPONENTES PRINCIPAIS

Observe que o *dataset* inicial, gerado na etapa descrita na Seção 4.1, é composto por uma matriz de 144 colunas (dimensões) por 21.455 linhas (observações). Para viabilizar o tempo de execução dos algoritmos de agrupamento e reduzir o risco de baixo desempenho devido à "maldição da dimensionalidade" (DUDA; HART; STORK, 2001), é necessário proceder com algum pré-processamento que diminua a dimensão do problema.

Dessa maneira, aplicou-se a reconhecida técnica de Análise de Componentes Principais. Consequentemente, o uso das características geradas, via PCA, reduz o número de colunas presentes na nova matriz de dados e, devido à restrição de descorrelação, também reduz o grau de redundância. Nesse sentido, é possível observar na Tabela 4.2 o acúmulo da variação dos dados (variância ou nível de energia) à medida que se considera mais componentes principais.

Tabela 4.2: Acúmulo da variação dos dados

PCA	PC1	PC2	PC3	PC4	PC5	PC6	PC7	PC8
Acumulada	49,6%	60,2%	68,4%	73,2%	77,0%	79,8%	82,0%	84,0%
PCA	PC9	PC10	PC11	PC12	PC13	PC14	PC15	PC16
Acumulada	85,8%	87,1%	88,2%	89,2%	90,1%	90,9%	91,7%	92,4%
PCA	PC17	PC18	PC19	PC20	PC21	PC22	PC23	PC24
Acumulada	93,2%	93,8%	94,4%	94,8%	95,2%	95,6%	95,9%	96,2%

As 20 primeiras componentes principais explicam 94,8% da variância dos dados. Elas foram selecionadas como características do vetor de entrada tanto para o algoritmo *K-Means* quanto para o *FCM*. Essas 20 primeiras foram selecionadas pelo fato de representar o mínimo de componentes principais capaz de explicar a variação dos dados sem perda significativa. No processo de validação do número de *clusters*, percebeu-se que trabalhar com um número inferior a 20 componentes principais afetaria a quantidade de *clusters* esperada para 144 dimensões. Desse modo, optou-se por 20 componentes principais por ser a quantidade mínima capaz de fornecer um resultado semelhante ao

de se utilizar todas as 144 dimensões. Vale destacar que essas 20 primeiras componentes principais são combinações lineares das 144 originais.

4.2.2 NÚMERO DE *CLUSTERS*

Conforme explicado anteriormente, uma das dificuldades em se aplicar os principais algoritmos de agrupamento, como o *K-Means* e o *FCM*, é determinar, antecipadamente, o número de *clusters* que deve ser gerado com base nas amostras de entrada. Em alguns casos, como o do presente trabalho, não se sabe, exatamente, o número correto de *clusters*, pois quer-se agrupar automaticamente os exemplares de acordo com o comportamento de execução apresentado (e não por classes geradas por rótulos de AV). Desse modo, para se chegar a um valor de parâmetro apropriado, foram usados dois métodos de validação: o método *cotovelo* e o *silhueta*.

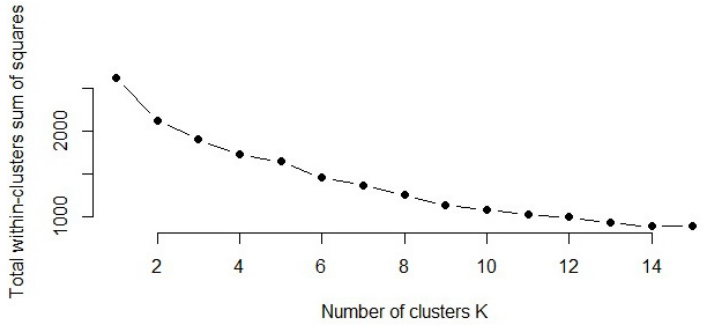

Figura 4.1: Aplicação do método cotovelo nas amostras de entrada

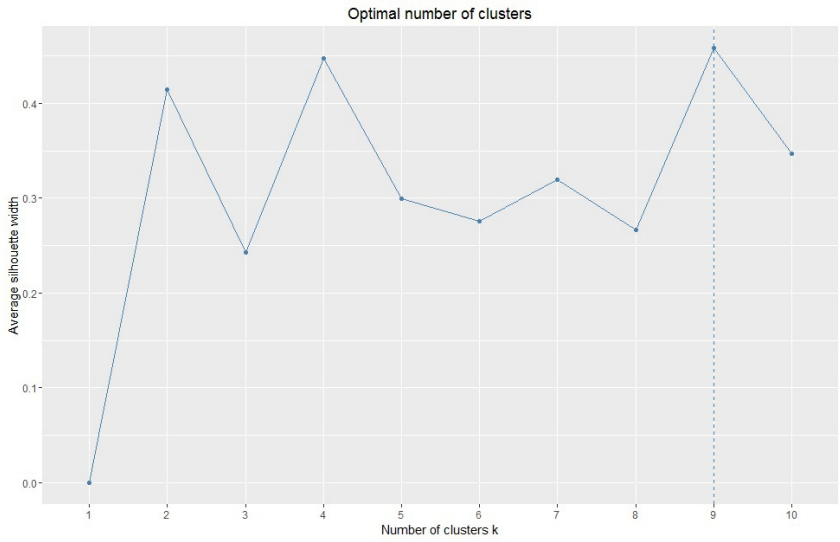

Figura 4.2: Aplicação do método silhueta nas amostras de entrada

Com o método cotovelo, não foi possível observar claramente o número de *clusters* necessários/adequados, de acordo com a Figura 4.1. Entretanto, a aplicação do método silhueta resultou na sugestão de definição de 9 *clusters*, conforme Figura 4.2. Logo, primeiramente, executou-se o algoritmo *K-Means* com $k = 9$, obtendo-se a distribuição das amostras conforme a Tabela 4.3.

Tabela 4.3: Distribuição das amostras entre os 9 *clusters*, após aplicação do *K-Means*.

Cluster	1	2	3	4	5	6	7	8	9
# Exemplares	5579	3526	740	1305	3656	409	1284	4765	191

Uma vez que o intuito é comparar o resultado dos algoritmos, o *FCM* também foi aplicado com a definição de 9 *clusters* como parâmetro de entrada. No caso do *FCM*, essa amostra não pertence, completamente, a um determinado *cluster*, os graus de pertinência indicam participação dela em cada *cluster*. Dessa forma, verificou-se em qual *cluster* o artefato possui maior pertinência, para fins de comparação com o *K-Means*. A distribuição dos exemplares resultante pode ser observada na Tabela 4.4.

Tabela 4.4: Distribuição das amostras entre os 9 *clusters*, após aplicação do *FCM*.

Cluster	1	2	3	4	5	6	7	8	9
# Exemplares	1945	2935	1830	399	1197	774	2277	5622	4476

5 RESULTADO DOS EXPERIMENTOS

Os resultados obtidos, por meio da aplicação dos dois algoritmos de aprendizado de máquina, foram analisados sob três aspectos: semelhança entre *clusters* produzidos, processo de rotulação e tempo de execução. A seguir, cada um dos aspectos supracitados é discutido.

5.1 SEMELHANÇA ENTRE *CLUSTERS*

Para a análise deste aspecto, foi desenvolvido um *script* em *Python* com a finalidade de comparar os *clusters* gerados como saída de cada algoritmo aplicado. Conforme mostrado na Tabela 5.1, pode-se observar o grau de coincidência dos exemplares para cada par de *clusters*, medido em porcentagem. Cinco *clusters* das duas técnicas apresentaram um elevado grau de semelhança, acima de 90%: por exemplo, o *cluster* K3 e C6 mostraram semelhança de 98%. O *cluster* 2 do *K-Means* (K2) mostrou a maior diferença, ao alocar 48% dos seus dados no *cluster* 1 do *FCM* (C1) e 45% no *cluster* 2 do *FCM* (C2). Com base nesses resultados, pode-se visualizar os agrupamentos semelhantes, conforme Tabela 5.1.

Tabela 5.1: Semelhança entre *clusters* produzidos por *K-Means* (K) e *FCM* (C).

Clusters	K1/C7	K1/C8	K2/C1	K2/C2	K3/C6	K4/C5
Semelhança	40%	60%	48%	45%	98%	91%
Clusters	K5/C2	K5/C8	K6/C4	K7/C3	K8/C9	K9/C3
Semelhança	37%	63%	95%	92%	93%	55%
Clusters	K9/C1					
Semelhança	40%					

Os métodos *K-Means* e *FCM* possuem os processos de inicialização, iteração e término bastante semelhantes. A diferença reside no emprego da função de pertinência por parte do método *FCM*. De fato, pode-se interpretar o *K-Means* como um caso especial do *FCM* para uma função de pertinência *crisp*, i.e. que retorna 1 se o ponto de dados é mais próximo do centroide ou 0, caso contrário. No caso do *FCM*, sabe-se

que a função de pertinência pode gerar valores entre 0 e 1. Ao analisar os *clusters* que apresentaram diferenças na Tabela 5.1, pode-se levantar a hipótese de que categorias de artefatos de comportamento mais complexo, à luz da função de pertinência *fuzzy*, levam a uma associação "diversificada" de *clusters*, o que não se alinha com os resultados equivalentes do *K-Means*.

5.2 PROCESSO DE ROTULAÇÃO

No momento de execução das amostras, o *Cuckoo* consulta a plataforma *online* do *VirusTotal*[12] de modo a verificar se o *malware* em questão já foi previamente enviado para detecção pelos antivírus disponíveis. Em caso positivo, o *VirusTotal* retorna um conjunto de rótulos, que corresponde ao resultado da varredura por vários AVs. Esse conjunto é armazenado como uma estrutura de dados no arquivo *JSON* referente ao relatório de cada exemplar. Como o cenário usual consiste em se obter rótulos distintos e/ou inconsistentes para um mesmo exemplar, elegeu-se o antivírus Avast para ser responsável pela atribuição de uma classe aos exemplares do conjunto deste experimento. A escolha do Avast deu-se pelo fato de este apresentar taxa de detecção elevada, de acordo com o VB100 (VirusBulletin, 2016).

Tabela 5.2: Exemplares de *malware* rotulados com Avast encontrados em *clusters* resultantes da aplicação do *K-Means* no *dataset*.

Cluster	K1	K2	K3	K4	K5	K6	K7	K8	K9	Total
Trojan	2118	847	83	287	1025	80	401	2170	27	7038
Worm	132	373	32	79	135	44	153	26	1	975
Spyware	243	182	15	45	98	9	20	130	0	742
Rootkit	72	13	7	5	27	4	10	7	0	145
PuP	258	627	437	384	432	99	53	341	70	2701
Genérico	1707	833	93	299	836	49	436	680	55	4988
Sem rótulo	1049	651	73	206	1103	124	211	1411	38	4866
Total	5579	3526	740	1305	3656	409	1284	4765	191	

Observa-se que o Avast, ao rotular as 21.455 amostras, distribuiu-as em cinco tipos de *malware* com escopo de comportamento bem definido—*Trojan*, *Worm*, *Spyware*, *Rootkit* e *Potentially Unwanted Program (PUP)*. O rótulo "Genérico"corresponde a nomes genéricos atribuídos pelo Avast (em geral detectados por procedimentos heurísticos),

[12] `https://www.virustotal.com`

enquanto que "Sem rótulo"refere-se aos artefatos que não foram identificados por este AV.

Ao analisar a Tabela 5.2, nota-se que 33% das amostram foram rotuladas como *Trojan* e 23% forma enquadradas como "Genérico", perfazendo um total de 55% de todas as amostras. Além disso, esses dois rótulos se espalharam por todos os *clusters*. Na verdade, os rótulos *Trojan* e "Genérico"são usados por muitos fornecedores de AV como rótulos guarda-chuva (MOHAISEN; ALRAWI, 2013), ou seja, cada um desses rótulos abrange mais de uma família de *malware*.

Nesta seção, apresenta-se uma solução para rotulação baseada em detecção por AV. No entanto, diferentemente do trabalho de (PIRSCOVEANU, 2015), utiliza-se a tabela de pertinência do *FCM* para auxiliar no processo: inicialmente, realiza-se a rotulação do agrupamento resultante do *K-Means* e, após isso, rotula-se com o *FCM*.

O rótulo atribuído a cada *cluster* advém da classe mais representativa encontrada dentro dele, de acordo com os resultados obtidos pelo Avast. Assim, o *cluster* K8 recebeu o rótulo *Trojan*, uma vez que possui 2170 exemplares classificados como tal.

Seguindo essa lógica, tem-se esta atribuição de rótulos: K8 - *Trojan*; K2 - *PuP*; K1 - *Spyware*; K5 - *Rootkit*; e K7 - *Worm*.

Nota-se que o tipo *Trojan* está presente em todos os agrupamentos e que cada rótulo só pode ser atribuído a um único *cluster*. Desse modo, por exemplo, ao atribuir o rótulo *Trojan* ao K8, este não pode ser concedido também ao K2, mesmo que nesse último *cluster* a quantidade desse tipo de artefato seja maioria. Nesse caso, opta-se pelo segundo mais representativo, no caso *PuP*. Ademais, como existem apenas cinco rótulos, quatro agrupamentos ficam sem rótulos (K3, K4, K6 e K9).

Nesse ponto, vale destacar que o trabalho de (PIRSCOVEANU, 2015), baseado em lógica convencional, também parou nesse ponto. Um dos objetivos desta dissertação é ir além nesse processo de rotulação, por meio da tabela de pertinência do FCM.

Para o método *FCM*, repete-se o mesmo processo, chegando aos seguintes rótulos para cada agrupamento, conforme Tabela 5.3: C9 = *Trojan*; C1 = *Worm*; C8 = *Spyware*; C6 = *PuP*; e C7 = *Rootkit*.

Tabela 5.3: *Malware* rotulados com Avast encontrados em *clusters* resultantes da aplicação do *FCM* no conjunto de exemplares.

Cluster	C1	C2	C3	C4	C5	C6	C7	C8	C9	Total
Trojan	393	928	463	78	263	89	874	1806	2144	7038
Worm	303	88	174	45	70	37	65	171	22	975
Spyware	101	110	72	8	40	19	127	164	101	742
Rootkit	8	21	11	3	5	7	27	58	5	145
PuP	381	425	220	96	351	443	131	400	254	2701
Genérico	360	755	584	47	281	101	571	1669	620	4988
Sem rótulo	399	608	306	122	187	78	482	1354	1330	4866
Total	1945	2935	1830	399	1197	774	2277	5622	4476	

Os agrupamentos C2, C3, C4 e C5 estão sem rótulo. No entanto, nesse caso, diferentemente do *K-Means*, pode-se tirar vantagem da tabela de pertinência. Para ilustrar essa ideia, foram selecionadas 10 amostras de cada um desses agrupamentos. Inicialmente, C2 com suas respectivas pertinências, conforme a Tabela 5.4.

Tabela 5.4: Tabela de pertinência de C2 com 10 exemplares, e seus dois *clusters* (C1 e C8) mais pertinentes/aderentes.

Amostras/Cluster	C1 (Worm)	C2	C8 (Spyware)
Exemplar 1	30%	32%	7,5%
Exemplar 2	10%	7,5%	7,6%
Exemplar 3	35%	43%	4,0%
Exemplar 4	20%	67%	4,1%
Exemplar 5	18%	28%	16%
Exemplar 6	1,8%	53%	42%
Exemplar 7	6,7%	86%	2,4%
Exemplar 8	14%	27%	26%
Exemplar 9	22%	44%	11%
Exemplar 10	5,3%	66%	22%

Na Tabela 5.4, C1 e C8 foram selecionados na medida em que apresentaram o maior grau de pertinência/aderência ao C2. Por exemplo, analisando o Exemplar 5, nota-se que sua pertinência ao C2 é 28%, seguido de C1 (18%) e C8 (16%). Desse modo, sugere-se o rótulo "Worm-Spyware"para o *cluster* C2.

Repetindo o processo para C3, tem-se a Tabela 5.5:

Tabela 5.5: Tabela de pertinência de C3 com 10 exemplares, e seus dois *clusters* (C1 e C9) mais pertinentes/aderentes.

Amostras/Cluster	C1 (Worm)	C3	C9 (Trojan)
Exemplar 1	8,6%	72%	13%
Exemplar 2	7,1%	71%	8,8%
Exemplar 3	7,9%	71%	13%
Exemplar 4	12,8%	71%	6,7%
Exemplar 5	8,7%	71%	7,0%
Exemplar 6	11%	71%	13%
Exemplar 7	8,5%	71%	11%
Exemplar 8	10%	70%	10%
Exemplar 9	10%	70%	6,3%
Exemplar 10	14%	70%	7,5%

Na Tabela 5.5, C1 e C9, foram selecionados por possuírem maior grau de pertinência/aderência ao C3. Desse modo, sugere-se o rótulo "Worm-Trojan"para o *cluster* C3.

Repetindo o processo para C4, tem-se a Tabela 5.6:

Tabela 5.6: Tabela de pertinência de C4 com 10 exemplares, e seus dois *clusters* (C3 e C6) mais pertinentes/aderentes.

Amostras/Cluster	C3 (Worm-Trojan)	C4	C6 (PuP)
Exemplar 1	8,8%	54%	10%
Exemplar 2	8,3%	54%	15%
Exemplar 3	8,2%	53%	4,5%
Exemplar 4	9,2%	52%	7,8%
Exemplar 5	12%	52%	5,9%
Exemplar 6	13%	51%	6,2%
Exemplar 7	13%	49%	6,4%
Exemplar 8	10%	49%	5,4%
Exemplar 9	9%	48%	17%
Exemplar 10	12%	48%	7,8%

Na Tabela 5.6, C3 e C6, foram selecionados por possuírem maior grau de pertinência/aderência ao C4. Desse modo, sugere-se o rótulo "PuP-Worm-Trojan"para o *cluster* C4.

Repetindo o processo para C5, tem-se a Tabela 5.7:

Tabela 5.7: Tabela de pertinência de C5 com 10 exemplares, e seus dois *clusters* (C1 e C6) mais pertinentes/aderentes.

Amostras/Cluster	C1 (Worm)	C5	C6 (PuP)
Exemplar 1	11%	21%	10%
Exemplar 2	11%	21%	10%
Exemplar 3	11%	21%	10%
Exemplar 4	11%	22%	9,9%
Exemplar 5	11%	22%	9,9%
Exemplar 6	11%	22%	10%
Exemplar 7	11%	23%	9,8%
Exemplar 8	11%	22%	10%
Exemplar 9	11%	24%	9,7%
Exemplar 10	11%	24%	9,7%

Na Tabela 5.7, C1 e C6 foram selecionados por possuírem maior grau de pertinência/aderência ao C5. Desse modo, sugere-se o rótulo "Worm-PuP"para o *cluster* C5.

Por fim, vale destacar que a rotulação, de um modo geral, fornece um subsídio importante ao analista de *malware*, ao especificar como rótulo um nome descritivo das características dos artefatos que ali se encontram. Em particular, o *FCM* permite verificar graus de relacionamento entre os *clusters*, ajudando a observar outros comportamentos da amostra. Por exemplo, ao rotular C2 com *"Worm-Spyware"*, o *FCM* informa ao analista que atividades maliciosas de *Worms* e *Spywares* devem ser monitoradas. Isso permite que a análise se concentre nas atividades normalmente executadas por essas espécies, tornando o exame mais direcionado ao real comportamento da amostra. Ademais, na atualidade, os artefatos normalmente executam atividades de mais de uma família, o que torna a modelagem por meio do *FCM* mais fidedigna.

5.3 TEMPO DE EXECUÇÃO E ESCALABILIDADE

Nos experimentos realizados, o algoritmo *K-Means* foi executado da seguinte forma: `kmeans(características, 9, nstart = 25)`, em que o parâmetro "características" representa os vetores obtidos do conjunto de amostras, o valor "9" corresponde ao número de *clusters* definidos para geração da saída e "nstart = 25" é o número de vezes diferentes que os *clusters* de partida são aleatoriamente gerados, a fim de que se possa escolher a melhor configuração para inicialização. Com relação ao *FCM*, este

foi executado da seguinte maneira: `cmeans(características, 9, iter.max = 100, dist = "euclidean", m = 1.5)`, em que os dois primeiros parâmetros são idênticos aos do *K-Means*, "iter.max = 100" é o número máximo de iterações para agrupamento, "dist = euclidean" é a métrica para calcular a distância entre os exemplares (no caso, distância Euclidiana), e "m = 1.5" se refere ao índice de *fuzzificação*. A Tabela 5.8 mostra a medida dos tempos para a aplicação de cada algoritmo.

Tabela 5.8: Tempo de execução medido para cada algoritmo.

Algoritmo	**Tempo (s)**
K-Means	21.1
FCM	19.6
Diferença	2.5

Cabe ressaltar que, apesar do *K-Means* possuir complexidade inferior ao *FCM*, especificamente nesse experimento, o parâmetro "nstart = 25", que possibilita uma aplicação mais robusta do *K-Means*, elevou, consideravelmente, seu tempo de execução.

Por fim, no caso de novas amostras a serem analisados, faz-se necessário reexecutar o método para agregar esses novos artefatos aos *clusters*. No entanto, todo o processo foi automatizado por meio de *scripts* em *Python* e R[13]

[13]Disponíveis em: `https://drive.google.com/drive/folders/0B09rUNJplNMLTGVrb0VvZ0tvaDA?usp=sharing`.

6 CONCLUSÕES

Esta dissertação propõe a aplicação de um método baseado em lógica *fuzzy* para alcançar uma forma adequada de agrupar exemplares de *malware* modernos. Pôde-se observar a vantagem do uso do algoritmo *Fuzzy C-Means (FCM)* em relação ao *K-Means*. No caso do *FCM*, as tabelas de pertinência permitem sugerir nomes aos grupo de exemplares de *clusters* e fornecem mais informações do que simples rótulos atribuídos por antivírus. Além disso, outro benefício de aplicação de lógica *fuzzy* em relação ao método de lógica *crisp* reside no fato de que os programas maliciosos não se limitam apenas a um comportamento específico de uma dada família, ou seja, podem pertencer a várias delas ao mesmo tempo. Portanto, a lógica *fuzzy*, apresentada nos resultados deste trabalho, modela, de forma mais fidedigna, o real comportamento malicioso exibido durante uma infecção.

No presente trabalho também se utilizaram duas técnicas para estimar o número de *clusters*, o método do cotovelo e o da silhueta, dado que o número de *clusters* não é conhecido a *priori*. No caso do método do cotovelo, não se conseguiu visualizar claramente a indicação do número de grupos. Desse modo, aplicou-se o método silhueta, no qual se chegou à sugestão de 9 *clusters*.

Por fim, observou-se a limitação de se utilizar antivírus no processo de rotulação, sugerindo-se um procedimento baseado nas características interna de cada grupo.

6.1 TRABALHOS FUTUROS

Como trabalhos futuros, pretende-se aprimorar o processo de rotulação de cada amostra e, consequentemente, dos *clusters* produzidos. Nos experimentos, percebeu-se que o principal objetivo de um fornecedor de AV é detectar códigos maliciosos, sendo o processo de rotulação uma prioridade secundária. Logo, AVs não possuem um processo de rotulação confiável, tornando necessário o desenvolvimento de um novo método sem uso de AV. De acordo os testes realizados neste trabalho, parece viável produzir rótulos baseados apenas nas características internas de cada amosta no *cluster*. Para isso, pode-se eleger as amostras mais representativas de cada *cluster* e estas devem ser

analisadas a fim de se descobrir seu comportamento e posterior rotulação do *cluster*. Nessa abordagem, o rótulo de cada *cluster* não seria mais baseado em AVs. Com relação à análise de novas amostras, pretende-se utilizar um método supervisionado, uma vez que se pode comparar melhor agrupamentos conhecidos *a priori*. Planeja-se também executar uma quantidade maior de exemplares para se verificar a formação de outros *clusters*, os quais podem apresentar novos comportamentos. Almeja-se também melhorar o processo de definição da quantidade de *clusters* e distribuição das amostras nos agrupamento, de modo a se ter mais dinamicidade na geração de modelos de classificação.

REFERÊNCIAS BIBLIOGRÁFICAS

ALTROCK, C. V. *Fuzzy logic and neurofuzzy applications in business and finance*. [S.l.]: Prentice-Hall, Inc., 1996.

ANDRADE, C. A. B.; MELLO, C. G. de; DUARTE, J. C. Malware automatic analysis. In: *2013 BRICS Congress on Computational Intelligence and 11th Brazilian Congress on Computational Intelligence*. [S.l.]: IEEE, 2013. p. 681 – 686.

BATSOS et al. Clustering and cartographic simplification of point data set. 2012.

BAZZI, C. L.; SOUZA, E. G. de; BETZEK, N. M. *Software para Definição de Unidades de Manejo: Teoria e Prática*. [S.l.: s.n.], 2015.

BEZDEK, J. C. *Pattern Recognition with Fuzzy Objective Function Algorithms*. 1. ed. Utah State University, Logan, Utah, USA: Kluwer Academic Publishers Norwell, MA, USA, 1981. ISBN 978-1-4757-0452-5.

BEZDEK, J. C.; EHRLICH, R.; FULL, W. FCM: The fuzzy c-means clustering algorithm. *Computers & Geosciences*, v. 10, n. 2–3, p. 191–203, 1984.

BORGES, V. R. P. Comparação entre as Técnicas de Agrupamento K-Means e Fuzzy C-Means para Segmentação de Imagens Coloridas. *XII Encontro Anual de Computação (EnAComp)*, 2010.

CERT, B. *Centro de Estudos, Respostas e Tratamento de Incidentes de Segurança no Brasil*. 2016. `http://www.cert.br/`. Acessado em agosto de 2016.

CHIU, C.-Y.; PARK, C. S. Fuzzy cash flow analysis using present worth criterion. *The Engineering Economist*, Taylor & Francis, v. 39, n. 2, p. 113–138, 1994.

COX, E. D. *Fuzzy logic for business and industry*. [S.l.]: Charles River Media, Inc., 1995.

DAMBALLA. *3% to 5% of Enterprise Assets Are Compromised by Bot-driven Targeted Attack Malware*. 3 2009. `http://www.prnewswire.com/news-releases/3-to-5-of-enterprise-assets-are-compromised-by-bot-driven-targeted-attack-malware-61634867.html`. Acessado em junho de 2016.

DAVATGAR, N.; NEISHABOURI, M.; SEPASKHAH, A. Delineation of site specific nutrient management zones for a paddy cultivated area based on soil fertility using fuzzy clustering. *Geoderma*, Elsevier, v. 173, p. 111–118, 2012.

DUARTE, J. C. et al. *Framework de Aprendizado de Máquina (FAMA)*. 2012. `https://code.google.com/archive/p/fama/`. Acessado em junho de 2016.

DUDA, R. O.; HART, P. E.; STORK, D. G. *Pattern Classification*. 2. ed. New York, USA: Wiley-Interscience, 2001.

DUNN, J. C. A Fuzzy Relative of the ISODATA Process and Its Use in Detecting Compact Well-Separated Clusters. *Journal of Cybernetics*, v. 3, n. 3, p. 32–57, 1974.

EDUREKA. *Number of clusters*. 2016. `http://www.edureka.co/blog/k-means-clustering/`. Acessado em setembro de 2016.

EGELE, M. et al. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)*, ACM, v. 44, n. 2, p. 6, 2012.

EILAM, E. *Reversing: secrets of reverse engineering*. [S.l.]: John Wiley & Sons, 2011.

ESTEVES, R. M.; RONG, C. Using Mahout for clustering Wikipedia's latest articles. *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, p. 565 – 569, 2011.

FAWCETT, T. An introduction to ROC analysis. *Elsevier Science Inc. New York, NY, USA*, v. 27, n. 8, p. 861—-874, 2006. ISSN 0167-8655.

FIRDAUSI, I. et al. Analysis of machine learning techniques used in behavior-based malware detection. In: *Proceedings of the 2010 Second International Conference on Advances in Computing, Control, and Telecommunication Technologies*. [S.l.: s.n.], 2010. (ACT'10), p. Pages 201 – 203.

FRIDGEN, J. J. et al. Management zone analyst (mza). *Agronomy Journal*, American Society of Agronomy, v. 96, n. 1, p. 100–108, 2004.

GANDOTRA, E.; BANSAL, D.; SOFAT, S. Malware analysis and classification: A survey. *Journal of Information Security*, Scientific Research Publishing, v. 2014, 2014.

GENTLEMAN, R.; IHAKA, R. *Project R*. 2014. `https://www.r-project.org/about.html`. Acessado em outubro de 2016.

GOUTTE, C. et al. On clustering fmri time series. *NeuroImage*, Elsevier, v. 9, n. 3, p. 298–310, 1999.

GROUP, M. L. *Weka*. 2016. http://www.cs.waikato.ac.nz/ml/weka/index.html. Acessado em outubro de 2016.

GUARNIERI, C. *Cuckoo Sandbox*. 2016. https://www.cuckoosandbox.org/. Acessado em junho de 2016.

HALKIDI, M.; BATISTAKIS, Y.; VAZIRGIANNIS, M. Cluster validity methods: part i. *ACM Sigmod Record*, ACM, v. 31, n. 2, p. 40–45, 2002.

HALL, M. et al. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, ACM, New York, NY, USA, v. 11, n. 1, p. 10–18, 2009. ISSN 1931-0145.

HAYKIN, S. Adaptive filter theory-3/e. 1996.

HU, X. *Large-Scale Malware Analysis, Detection, and Signature Generation*. Tese (Doutorado) — The University of Michigan, 2011.

HUANG, H.-D. et al. Fuzzy markup language for malware behavioral analysis. In: *On the Power of Fuzzy Markup Language*. [S.l.]: Springer, 2013. p. 113–132.

HYVARINEN, A.; KARHUNEN, J. Oja. independent component analysis. *John Wiley&Sonr*, 2001.

Intel Security. *Relatório do McAfee Labs sobre ameaças*. 2016. https://www.partner.securecomputing.com/br/resources/misc/infographic-threats-report-dec-2016.pdf. Acessado em janeiro de 2017.

ISECLAB. *Anubis - Analyzing Unknown Binaries*. 2015. http://analysis.iseclab.org/. Acessado em junho de 2015.

JAIN, A. K.; DUBES, R. C. *Algorithms for clustering data*. [S.l.]: Prentice-Hall, Inc., 1988.

JAMES, G. et al. *An introduction to statistical learning*. [S.l.]: Springer, 2013.

JANÉ, D. d. A. Uma introdução ao estudo da lógica fuzzy. *Hórus*, v. 2, p. 1–16, 2004.

KAUFMANN, A.; GUPTA, M. M. *Fuzzy mathematical models in engineering and management science*. [S.l.]: Elsevier Science Inc., 1988.

KETCHEN, D. J.; SHOOK, C. L. The application of cluster analysis in strategic management research: an analysis and critique. *Strategic management journal*, JSTOR, v. 17, n. 6, p. 441–458, 1996.

KING, S. T.; CHEN, P. M. Subvirt: Implementing malware with virtual machines. In: IEEE. *2006 IEEE Symposium on Security and Privacy (S&P'06)*. [S.l.], 2006. p. 14–pp.

MACQUEEN, J. Some Methods for Classification and Analysis of Multivariate Observations. *5-th Berkeley Symposium on Mathematical Statistics and Probability*, p. 281 – 297, 1967.

MALIN, C. H.; AQUILINA, J. M.; CASEY, E. *Malware Forensics Field Guide for Windows Systems: Digital Forensics Field Guides*. [S.l.]: Elsevier, 2011.

MANGIALARDO, R. J. *Integrando as Análises Estática e Dinâmica na Identificação de Malwares Utilizando Aprendizado de Máquina*. Dissertação (Mestrado) — Mestrado em Sistemas e Computação, Instituto Militar de Engenharia, Rio de Janeiro, 2015.

MARTINS, C. A. *Uma abordagem para pré-processamento de dados textuais em algoritmos de aprendizado. 2003. 90 f.* Tese (Doutorado) — Tese (Doutorado em Ciências da Computação e Matemática Computacional)–Instituto de Ciências Matemáticas e de Computação–ICMCUSP, USP-São Carlos, 2003.

METZ, J. *Interpretaçao de clusters gerados por algoritmos de clustering hierárquico*. Tese (Doutorado) — Universidade de São Paulo, 2006.

MITCHELL, T. M. *Machine learning*. [S.l.: s.n.], 1997.

MOHAISEN, A.; ALRAWI, O. Unveiling zeus: automated classification of malware samples. In: ACM. *Proceedings of the 22nd International Conference on World Wide Web*. [S.l.], 2013. p. 829–832.

MONARD, M. C.; BARANAUSKAS, J. A. Conceitos sobre aprendizado de máquina -. *Sistemas Inteligentes-Fundamentos e Aplicações*, v. 1, n. 1, 2003.

MOSER, A.; KRUEGEL, C.; KIRDA, E. Limits of static analysis for malware detection. In: IEEE. *Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual*. [S.l.], 2007. p. 421–430.

NASSIF, L. F. d. C. Técnicas de agrupamento de textos aplicadas à computação forense. 2012.

ODEH, I.; CHITTLEBOROUGH, D.; MCBRATNEY, A. Soil pattern recognition with fuzzy-c-means: application to classification and soil-landform interrelationships. *Soil Science Society of America Journal*, Soil Science Society of America, v. 56, n. 2, p. 505–516, 1992.

OMATU, S. et al. *Distributed computing and artificial intelligence*. [S.l.]: Springer, 2014.

PINHO, A. F. *Uma contribuição para a resolução de problemas de programação de operações, em sistemas de produção intermitentes flow-shop: a consideração de incertezas*. Tese (Doutorado) — Dissertação de Mestrado Departamento de Produção, UNIFEI, 1999.

PIRSCOVEANU, R.-S. *Clustering Analysis of Malware Behavior*. Dissertação (Mestrado) — Institute of Electronic Systems Department of Communication Technology at Aalborg University, Denmark, 2015.

PROVATAKI, A.; KATOS, V. Differential malware forensics. *Digital Investigation: The International Journal of Digital Forensics & Incident Response*, v. 10, n. 4, p. Pages 311– 322, 12 2013.

ROUSSEEUW, P. J. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, v. 20, p. 53–65, 1987.

SALEHI, Z.; GHIASI, M.; SAMI, A. A Miner for Malware Detection Based on API Function Calls and Their Arguments. In: SALEHI, Z. (Ed.). *Artificial Intelligence and Signal Processing (AISP)*. [S.l.]: IEEE, 2012. p. 563 – 568.

SAMI, A. et al. Malware detection based on mining api calls. In: *Proceedings of the 2010 ACM Symposium on Applied Computing*. [S.l.]: ACM New York, NY, USA, 2010. (SAC'10), p. 1020 – 1025.

SIKORSKI, M.; HONIG, A. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. 1st. ed. San Francisco, CA, USA: No Starch Press, 2012. ISBN 1593272901, 9781593272906.

SWIESKOWSKI, P.; KUZINS, S. *Ninite*. 2016. `https://ninite.com/`. Acessado em fevereiro de 2016.

SZOR, P. *The art of computer virus research and defense*. [S.l.]: Pearson Education, 2005.

VirusBulletin. *VB100*. 2016. `https://www.virusbulletin.com/testing/vb100/`. Acessado em fevereiro de 2016.

YEN, J.; LANGARI, R.; ZADEH, L. A. *Industrial applications of fuzzy logic and intelligent systems*. [S.l.]: IEEE press, 1995.

ZADEH, L. A. Fuzzy sets. *Information and control*, Elsevier, v. 8, n. 3, p. 338–353, 1965.

ZHANG, Y. et al. Fuzzy neural network for malware detect. In: IEEE. *Intelligent System Design and Engineering Application (ISDEA), 2010 International Conference on*. [S.l.], 2010. v. 1, p. 780–783.

ZUBEN, I.-P. F. J. V.; ATTUX, R. R. Análise de componentes independentes (ica). 2010.

APÊNDICES

Apêndice 1 - Código fonte em *Python* para realizar a leitura dos relatório JSON gerados pelo Cuckoo Sandboxing

```
# alguns trechos de codigo relacionados a leitura dos json
#foram reaproveitados de  Mike Schladt - 2015
#(https://github.com/schladt/malware-feature-vectors)

import argparse
import requests
import json
import sys
import csv


headers = ['NtOpenKeyEx' , 'DeleteFileA' , 'NtDeleteKey',
'NtQueryValueKey', 'CreateThread', 'NtSuspendThread',
'LdrGetProcedureAddress', 'RegDeleteValueW', 'accept',
'GetSystemMetrics',  'NtEnumerateValueKey', 'InternetOpenUrlA'
    ,
'NtQueryKey', 'select', 'GetAddrInfoW',
'RegCloseKey', 'WriteConsoleA', 'InternetReadFile',
'RegOpenKeyExW', 'ReadProcessMemory', 'NtSetContextThread',
'RegQueryValueExA', 'ExitThread', 'recvfrom',
'NtQueryInformationFile', 'NtSetValueKey', 'CreateRemoteThread
    ',
'RegQueryValueExW', 'WriteProcessMemory', 'ShellExecuteExW',
'NtDelayExecution', 'NtResumeThread', 'NtCreateNamedPipeFile',
'NtSetInformationFile', 'recv', 'WSASocketW',
'NtReadFile', 'socket', 'InternetConnectA',
'NtCreateFile', 'NtOpenDirectoryObject', 'StartServiceA',
'LdrLoadDll', 'connect', 'HttpOpenRequestA',
'RegOpenKeyExA', 'DeleteFileW', 'HttpSendRequestA',
'NtReadVirtualMemory', 'closesocket', 'NtWriteVirtualMemory',
'NtEnumerateKey', 'anomaly', 'InternetOpenUrlW',
'NtOpenFile', 'VirtualProtectEx', 'CreateServiceA',
'ZwMapViewOfSection', 'RegDeleteValueA', 'InternetConnectW',
'NtDeviceIoControlFile', 'ExitProcess', 'HttpOpenRequestW',
```

```
'NtOpenKey', 'setsockopt', 'HttpSendRequestW',
'FindWindowA', 'FindWindowExW', 'sendto',
'FindFirstFileExW', 'ioctlsocket', 'MoveFileWithProgressW',
'NtOpenMutant', 'CopyFileA', 'CopyFileW',
'LdrGetDllHandle', 'gethostname', 'listen',
'NtWriteFile', 'OpenSCManagerA', 'WSASend',
'RegEnumValueW', 'WSAStartup', 'RegDeleteKeyW',
'NtQueryDirectoryFile', 'OpenServiceA', 'RemoveDirectoryW',
'NtCreateSection', 'send', 'NtSaveKey',
'GetCursorPos', 'OpenSCManagerW', 'NtTerminateProcess',
'NtFreeVirtualMemory', 'RegDeleteKeyA', 'NtTerminateThread',
'NtProtectVirtualMemory', 'OpenServiceW', 'StartServiceW',
'FindWindowExA', 'NtOpenThread', 'CopyFileExW',
'RegSetValueExA', 'RegQueryInfoKeyA', 'WSASocketA',
'RegCreateKeyExW', 'SetWindowsHookExW', 'system',
'RegEnumKeyW', 'getaddrinfo', 'DeleteService',
'NtOpenSection', 'NtCreateThreadEx', 'DnsQuery_A',
'CreateDirectoryW', 'InternetCloseHandle', 'ExitWindowsEx',
'RegCreateKeyExA', 'bind', 'WSARecvFrom',
'CreateProcessInternalW', 'RemoveDirectoryA', '
  CreateDirectoryExW',
'RegEnumValueA', 'UnhookWindowsHookEx', 'NtDeleteValueKey',
'RegEnumKeyExA', 'LookupPrivilegeValueW', 'CreateServiceW',
'NtCreateKey', 'SetWindowsHookExA', 'FindFirstFileExA',
'FindWindowW', 'URLDownloadToFileW', 'RtlCreateUserThread',
'RegSetValueExW', 'WSARecv', 'InternetWriteFile',
'WriteConsoleW', 'shutdown', 'NtLoadKey',
'RegEnumKeyExW', 'NtGetContextThread', 'WSASendTo',
'DeviceIoControl', 'InternetOpenA', 'NtMakeTemporaryObject',
'RegQueryInfoKeyW', 'ControlService', '
  NtQueryMultipleValueKey',
'NtCreateMutant', 'InternetOpenW', 'NtSaveKeyEx',
'Downloads', 'Rotulo', 'Tipo',
'Chamadas', 'MD5']

CUCKOO_REST_HOST = 'http://localhost:8090/'
```

```
def create_feature_vector2(json_report) :
    """
    Process cuckoo v1.2 report to extract  compatible tags
    Input : report : json object : cuckoo report as json
       object
    Output : tags : set : set of tags
    """
    #create container for features
    features = {}



    if 'behavior' in json_report :
#         if 'summary' in json_report['behavior'] :
#             for key, value in json_report['behavior']['
   summary'].iteritems() :
#                 features['behavior_summary_'+key] = len(
   value)

            #api stats
        if 'apistats' in json_report['behavior'] :
            for item in json_report['behavior']['apistats'] :
                for key, value in json_report['behavior']['
                   apistats'][item].iteritems() :
                    print(key)
                    print(value)
                    print(json_report['target']['file']['md5'
                       ])
                    if 'behavior_apistats_'+key in features :
                        features[key] = features[key] + value
                    else :
                        features[key] = value

    return features

def cuckoo_task_list():
    """
```

```
    Returns Cucko task list
    Input : NONE
    Output : cuckoo_tasks :iterable of Cuckoo tasks
    """

    cuckoo_tasks = []
    #get status of active tasks
    cuckoo_rest_host = CUCKOO_REST_HOST
    request_url = "{0}tasks/list".format(cuckoo_rest_host)
    response = requests.get(request_url).json()['tasks'];
    for task in response :
        cuckoo_task = {}
        cuckoo_task['task_id'] = task['id']
        cuckoo_task['md5'] = task['sample']['md5']
        cuckoo_task['status'] = task['status']

        cuckoo_tasks.append(cuckoo_task)

    return cuckoo_tasks


def get_report(task_id) :
    """
    Uses cuckoo REST api to get JSON output of reported file
    INPUT: task_id : int : reported cuckoo task
    OUTPUT : report : json object of cuckoo report
    """
    cuckoo_rest_host = CUCKOO_REST_HOST
    request_url = "{0}tasks/report/{1}".format(
       cuckoo_rest_host, task_id)
    response = requests.get(request_url)
    report = response.json()

    return report

def delete_task(task_id) :
    """
```

```
    Removes provided task_id from the Cuckoo database via API
    INPUT : task_id : int : cuckoo task id
    OUTPUT : bool : True if successful, False if not
    """
    cuckoo_rest_host = CUCKOO_REST_HOST
    request_url = "{0}tasks/delete/{1}".format(
        cuckoo_rest_host, task_id)
    response = requests.get(request_url)
    if response.status_code == 200 :
        return True
    else :
        return False


#list to store each sample's feature vector
samples = []
cuckoo_tasks = []


#get all of the task id's from cuckoo
cuckoo_tasks = cuckoo_task_list()



with open('output.csv','w') as out:
    # abre o arquivo csv que sera gerado
    output = csv.writer(out)
    output.writerow(headers)

    #get reports for each task id
    for task in cuckoo_tasks :

        rows = [[0 for col in xrange(158)]]
        numChamadas = 0

        if task['status'] == 'reported' :
            try :
```

```
report = get_report(task['task_id'])

if 'behavior' in report :

    #api stats
    if 'apistats' in report['behavior'] :
        for item in report['behavior']['
           apistats'] :
        # print item

            for key, value in report['behavior
               ']['apistats'][item].iteritems
               () :
               # print key
               # print value
                for x in xrange(0, 153):


                        if (key == headers[x])
                           :
                            if rows[0][x] !=
                               0:
                                rows[0][x]=
                                   rows[0][x]
                                   + value
                                numChamadas=
                                   numChamadas
                                    + value
                            else:
                                rows[0][x]=
                                   value
                                numChamadas=
                                   numChamadas
                                    + value

        if 'dropped' in report :
```

```
        rows[0][153]= len(report['dropped'
            ])

    if 'target' in report:
        if 'file' in report['target']:
            if 'type' in report['target'][
                'file']:
                rows[0][155]= report['
                    target']['file']['type'
                    ]

            if 'md5' in report['target']['
                file']:
                rows[0][157]= report['
                    target']['file']['md5']

    if 'virustotal'in report:
        if 'scans'in report['virustotal']:
            if 'Avast' in report['
                virustotal']['scans']:
                rows[0][154]= report['
                    virustotal']['scans']['
                    Avast']['result']



    rows[0][156]= numChamadas


    # numero de chamadas inferiores a 50
        eh desconsiderado. Isso se deve ao
        fato que
    # um numero reduzido de chamdas
        invabiliza o processo de
        classificacao.
    if numChamadas >= 50 :
        output.writerows(rows)
```

```
except Exception as e :
    sys.stderr.write("Erro ao processar a  TASK
        {0}".format(task['task_id']))
```

Apêndice 2 - Código fonte em *Python* para realizar remoção de linhas ou colunas que não possuem dados

```
import pandas as pd
import csv
from dpkt.asn1 import NULL
from pandas.core.index import Index


lista=[]
linha=[]

with open('/root/workspace/FormataCSV/src/output.csv') as f:
    # todo o arquivo de entrada e colocado em um vetor. A
       linha um e colcada na posicao lis[0]
    # e assim por diante.
    lis = [x.split(',') for x in f]
  #  print lis;
count=0
acessada=False
numColunas=0;
numLinhas=0;

numAtributos= 153;
contador=0;
# a variavel x recebe cada coluna do csv. Ou seja, a coluna
   vira uma linha e cada elemento
# pode ser acessado como x[i]. O objetivo desse tranformacao e
    facilitar o acesso
for x in zip(*lis):


        if contador < numAtributos:
        # a varivael y recebe cada elemento de x. Inicialmente
           , o nome da coluna, depois o elemento na primeira
        # linha e assim sucessivamente.
       # print x;
            for y in x:
```

```
            #     print y;

                if count==0:
                    nomeFuncao=y
                  #  print nomeFuncao;
                    count=count+1
                elif int(y)!=0:

                    acessada=True


            if acessada==True:
            #     nomeFuncao=nomeFuncao.replace ("\n", "")
                nomeFuncao=nomeFuncao.replace('\r', '')
                nomeFuncao=nomeFuncao.replace('\n', '')
                lista.append(nomeFuncao)

              #  print nomeFuncao


        if contador >= numAtributos:
            for z in x:
            #     print y;

                if count==0:
                    z=z.replace('\r', '')
                    z=z.replace('\n', '')
                    lista.append(z)
                    count=count+1


        contador= contador + 1;
        count=0
        acessada=False
#str.replace("\n", "")
# realiza a leitura do arquivo de entrada
```

```
df = pd.read_csv("output.csv")


print lista
# mantem no arquivo de entrada apenas as colunas em lista
new_df=df[lista]
numLinhas= new_df.shape[0];
numColunas= new_df.shape[1];

#print numLinhas;
#print numColunas;


acessada=False
for x in range(0, numLinhas):
    for y in range(0, numColunas):

        try:

            if  new_df.iloc[x][y] != 0:

                acessada=True;

        except Exception:
            pass

    if acessada == True:
        linha.append(x);

    acessada=False
df = new_df.ix[linha]
print df


df.to_csv("caracteristicas.csv", header=True, sep=',',
   enconding=None, line_terminator='\n', index=False)
```

Apêndice 3 - Código fonte em *Python* para comparar *clusters* gerados pelos*K-Means* e *FCM*

```
import csv

headers = ['Clusters', 'Intersecao']

with open('comparativo.csv','w') as out:
    # abre o arquivo csv que sera gerado
    output = csv.writer(out)
    output.writerow(headers)
    rows = [[0 for col in xrange(2)]]
    number_rowsk=0;
    number_rowsc=0;
    for k in xrange(1,2):
        number_rowsk=0;

        with open('/media/sf_Sistemas_Operacionais/Clusters-K-
            means/k-means-c' + str(k) + '.csv', 'rb') as
            clusterk:

                readerk = csv.DictReader(clusterk)
                count=0;
                for rowk in readerk:
                    number_rowsk = number_rowsk + 1;

                    for c in xrange(1,11):
                        number_rowsc =0;

                        with open('/media/
                            sf_Sistemas_Operacionais/Clusters-
                            FCM/c-means-c' + str(c) + '.csv', '
                            rb') as clusterc:
                            readerc = csv.DictReader(clusterc)
                            for rowc in readerc:

                                number_rowsc = number_rowsc +
                                    1;
```

```
                if rowk['MD5']!= str(0) and
                    rowc['MD5']!= (0):
                    if rowk['MD5'] == rowc['
                        MD5']:

                            # rows[0][x]=rows
                                [0][x]
                        count = count +1;
            if count != 0:
                rows[0][0] = 'K'+ str(k) + '('
                    + str(number_rowsk) + ')' +
                     '/' + 'C' + str(c) + '(' +
                     str(number_rowsc) + ')';
                rows[0][1] = count;
                output.writerows(rows)

#   print 'cluster n ' + str(k)+ '
    possui as seguintes coincidencicas
    = ' + str(count);
```

Apêndice 4 - Código fonte em *Python* para contar o número de amostras rotuladas de cada grupo nos *clusters* gerados pelos *K-Means* e *FCM*

```
import pandas as pd

df = pd.read_csv("c-cluster9.csv")

new_df=df['Rotulo'];
numLinhas= new_df.shape[0];



contTrj=0;
contWrm=0;
contSpy=0;
contPuP=0;
contRtk=0;
contDrp=0;
contCryp=0;
contVazio=0;
srt1="";

with open('/root/workspace/FormataCSV/src/c-cluster9.csv') as
    f:
     # todo o arquivo de entrada e colocado em um vetor. A
        linha um e colcada na posicao lis[0]
     # e assim por diante.e
     lis = [x.split(',') for x in f]
count=0
contador=0;
for x in zip(*lis):


          if contador < numLinhas:
          # a varivael y recebe cada elemento de x. Inicialmente
              , o nome da coluna, depois o elemento na primeira
          # linha e assim sucessivamente.
         # print x;
               for y in x:
```

```
                    if count==0:
                        nomeFuncao=y
                        count=count+1
                    elif int(y.find("Trj"))!= -1 or int(y.find("
                        Trojan"))!= -1: #or int(y.find("Drp"))!= -1
                        or int(y.find("Drop"))!= -1 or int(y.find
                        ("Cryp"))!= -1 or int(y.find("Cryp"))!= -1:
                        contTrj = contTrj + 1;
                    elif int(y.find("Wrm"))!= -1 or int(y.find("
                        Worm"))!= -1:
                        contWrm = contWrm + 1;
                    elif int(y.find("Spy"))!= -1 or int(y.find("
                        Spyware"))!= -1 or int(y.find("Adw"))!= -1
                        or int(y.find("Adware"))!= -1:
                        contSpy = contSpy + 1;
                    elif int(y.find("PUP"))!= -1:
                        contPuP = contPuP + 1;
                    elif int(y.find("Rtk"))!= -1 or int(y.find("
                        Rootkit"))!= -1:
                        contRtk = contRtk + 1;
                #   elif y == "\n" or int(y.find("0\n"))!= -1:
                    elif y == "\n" or y == "0\n":
                        contVazio = contVazio + 1;
            contador = contador +1;
print "Numero de Trojans = " + str(contTrj);
print "Numero de Worms = " + str(contWrm);
print "Numero de Spywares = " + str(contSpy);
print "Numero de PuP = " + str(contPuP);
print "Numero de Rootkits = " + str(contRtk);
print "Numero de nao rotulados = " + str(contVazio);
print  "Outros = " + str(numLinhas-contTrj-contWrm-contSpy-
    contPuP-contRtk-contVazio);
print "Total de amostras = " + str(numLinhas);
```

Apêndice 5 - Código fonte em R para validar o número de *clusters* pelo métodos cotovelo e silhueta

```
# total within-cluster sum of square (WSS)

set.seed(123)
# Compute and plot wss for k = 2 to k = 15
k.max <- 15 # Maximal number of clusters
data <- M1
wss <- sapply(1:k.max,
              function(k){kmeans(data, k, nstart=10 )$tot.
                  withinss})

plot(1:k.max, wss,
     type="b", pch = 19, frame = FALSE,
     xlab="Number of clusters K",
     ylab="Total within-clusters sum of squares")

#-------------------------------------------------------------------
# silhouette width

set.seed(123)
k.max <- 15
data <- M1
sil <- rep(0, k.max)

# Compute the average silhouette width for
# k = 2 to k = 15
for(i in 2:k.max){
  km.res <- kmeans(data, centers = i, nstart = 25)
  ss <- silhouette(km.res$cluster, dist(data))
  sil[i] <- mean(ss[, 3])
}

# Plot the  average silhouette width
plot(1:k.max, sil, type = "b", pch = 19, frame = FALSE, xlab =
    "Number of clusters k")
```

```
#abline(v = which.max(sil), lty = 2)

fviz_nbclust(M1, kmeans, method = "wss")
```

Apêndice 6 - Código fonte em R da aplica do método *K-Means*

```
library(factoextra)
library(cluster)
library(NbClust)

set.seed(123)
caracteristicas4500Completo <- read.csv("D:/Mestrado - UNB/
    Sistemas Operacionais/caracteristicas15000.csv")

caracteristicas4500=subset(caracteristicas4500Completo, select
    = -c(Downloads:MD5))

M1 = sweep(caracteristicas4500,1,rowSums(caracteristicas4500),
    "/")
# pr.out =prcomp (M1, center = FALSE, scale =FALSE)
# pc16= pr.out$x[,1:16]
# summary(pr.out)

start.time <- Sys.time()
km.res <- kmeans(M1, 10, nstart = 25)
end.time <- Sys.time()
time.taken <- end.time - start.time
time.taken


table(km.res$cluster)

c1 <- caracteristicas4500Completo[which(km.res$cluster == 1),]
c2 <- caracteristicas4500Completo[which(km.res$cluster == 2),]
c3 <- caracteristicas4500Completo[which(km.res$cluster == 3),]
c4 <- caracteristicas4500Completo[which(km.res$cluster == 4),]
c5 <- caracteristicas4500Completo[which(km.res$cluster == 5),]
c6 <- caracteristicas4500Completo[which(km.res$cluster == 6),]
c7 <- caracteristicas4500Completo[which(km.res$cluster == 7),]
c8 <- caracteristicas4500Completo[which(km.res$cluster == 8),]
c9 <- caracteristicas4500Completo[which(km.res$cluster == 9),]
```

```
c10 <- caracteristicas4500Completo[which(km.res$cluster == 10)
    ,]

write.csv(c1, file = "D:/Mestrado - UNB/k-means-c1.csv")
write.csv(c2, file = "D:/Mestrado - UNB/k-means-c2.csv")
write.csv(c3, file = "D:/Mestrado - UNB/k-means-c3.csv")
write.csv(c4, file = "D:/Mestrado - UNB/k-means-c4.csv")
write.csv(c5, file = "D:/Mestrado - UNB/k-means-c5.csv")
write.csv(c6, file = "D:/Mestrado - UNB/k-means-c6.csv")
write.csv(c7, file = "D:/Mestrado - UNB/k-means-c7.csv")
write.csv(c8, file = "D:/Mestrado - UNB/k-means-c8.csv")
write.csv(c9, file = "D:/Mestrado - UNB/k-means-c9.csv")
write.csv(c10, file = "D:/Mestrado - UNB/k-means-c10.csv")
```

Apêndice 7 - Código fonte em R da aplicação do método *FCM*

```
library("e1071")
library(factoextra)
library(cluster)
library(NbClust)

set.seed(123)
caracteristicas4500Completo <- read.csv("D:/Mestrado - UNB/
    Sistemas Operacionais/caracteristicas15000.csv")

caracteristicas4500=subset(caracteristicas4500Completo, select
    = -c(Downloads:MD5))

M1 = sweep(caracteristicas4500,1,rowSums(caracteristicas4500),
    "/")

start.time <- Sys.time()
cm <- cmeans(M1, 10, iter.max = 100, dist = "euclidean", m =
    1.5)
end.time <- Sys.time()
time.taken <- end.time - start.time
time.taken

table(cm$cluster)

c1 <- caracteristicas4500Completo[which(cm$cluster == 1),]
c2 <- caracteristicas4500Completo[which(cm$cluster == 2),]
c3 <- caracteristicas4500Completo[which(cm$cluster == 3),]
c4 <- caracteristicas4500Completo[which(cm$cluster == 4),]
c5 <- caracteristicas4500Completo[which(cm$cluster == 5),]
c6 <- caracteristicas4500Completo[which(cm$cluster == 6),]
c7 <- caracteristicas4500Completo[which(cm$cluster == 7),]
c8 <- caracteristicas4500Completo[which(cm$cluster == 8),]
c9 <- caracteristicas4500Completo[which(cm$cluster == 9),]
c10 <- caracteristicas4500Completo[which(cm$cluster == 10),]

member <- cm$membership[which(cm$cluster == 8),]
```

```
write.csv(member, "D:/Mestrado - UNB/Sistemas Operacionais/
    membership.csv")

write.csv(c1, file = "D:/Mestrado - UNB/c-means-c1.csv")
write.csv(c2, file = "D:/Mestrado - UNB/c-means-c2.csv")
write.csv(c3, file = "D:/Mestrado - UNB/c-means-c3.csv")
write.csv(c4, file = "D:/Mestrado - UNB/c-means-c4.csv")
write.csv(c5, file = "D:/Mestrado - UNB/c-means-c5.csv")
write.csv(c6, file = "D:/Mestrado - UNB/c-means-c6.csv")
write.csv(c7, file = "D:/Mestrado - UNB/c-means-c7.csv")
write.csv(c8, file = "D:/Mestrado - UNB/c-means-c8.csv")
write.csv(c9, file = "D:/Mestrado - UNB/c-means-c9.csv")
write.csv(c10, file = "D:/Mestrado - UNB/c-means-c10.csv")
```