MINISTÉRIO DA DEFESA
EXÉRCITO BRASILEIRO
DEPARTAMENTO DE CIÊNCIA E TECNOLOGIA
INSTITUTO MILITAR DE ENGENHARIA
CURSO DE MESTRADO EM SISTEMAS E COMPUTAÇÃO

CÉSAR AUGUSTO BORGES DE ANDRADE

ANÁLISE AUTOMÁTICA DE MALWARES UTILIZANDO AS TÉCNICAS DE SANDBOX E APRENDIZADO DE MÁQUINA

Rio de Janeiro
2013

INSTITUTO MILITAR DE ENGENHARIA

CÉSAR AUGUSTO BORGES DE ANDRADE

ANÁLISE AUTOMÁTICA DE MALWARES UTILIZANDO AS TÉCNICAS DE SANDBOX E APRENDIZADO DE MÁQUINA

Dissertação de Mestrado apresentada ao Curso de Mestrado em Sistemas e Computação do Instituto Militar de Engenharia, como requisito parcial para obtenção do título de Mestre em Sistemas e Computação.

Orientador: Prof. Claudio Gomes de Mello - D.C.
Co-orientador: Prof. Julio Cesar Duarte - D.C.

Rio de Janeiro
2013

INSTITUTO MILITAR DE ENGENHARIA
Praça General Tibúrcio, 80-Praia Vermelha
Rio de Janeiro-RJ CEP 22290-270

004.1 Andrade, César Augusto Borges de
A553a Análise automática de malwares utilizando as técnicas de sandbox e aprendizado de máquina/ César Augusto Borges de Andrade. - Rio de Janeiro: Instituto Militar de Engenharia, 2013.

81 p.: il., tab.

Dissertação (mestrado) – Instituto Militar de Engenharia – Rio de Janeiro, 2013.

1. Análise de Malware. 2. Aprendizado de Máquina. I. Título. II. Instituto Militar de Engenharia.

CDD 004.1

INSTITUTO MILITAR DE ENGENHARIA

CÉSAR AUGUSTO BORGES DE ANDRADE

ANÁLISE AUTOMÁTICA DE MALWARES UTILIZANDO AS TÉCNICAS DE SANDBOX E APRENDIZADO DE MÁQUINA

Dissertação de Mestrado apresentada ao Curso de Mestrado em Sistemas e Computação do Instituto Militar de Engenharia, como requisito parcial para obtenção do título de Mestre em Sistemas e Computação.

Orientador: Prof. Claudio Gomes de Mello - D.C.

Co-orientador: Prof. Julio Cesar Duarte - D.C.

Aprovada em 31 de janeiro de 2013 pela seguinte Banca Examinadora:

Prof. Claudio Gomes de Mello - D.C. do IME - Presidente

Prof. Julio Cesar Duarte - D.C. do IME

Prof. Ruy Luiz Milidiú - Ph.D. da PUC-Rio

Prof. Anderson Fernandes Pereira dos Santos - D.C. do IME

Rio de Janeiro
2013

Dedico esta a João Batista, meu pai.

AGRADECIMENTOS

Agradeço a todas as pessoas que contribuíram com o desenvolvimento desta dissertação de mestrado, por meio de críticas, ideias, apoio, incentivo ou qualquer outra forma de auxílio. Em especial, desejo agradecer à minha esposa Adriana Cristina Moreira Borges de Andrade, pelo apoio, ao meu filho Heitor Moreira Borges de Andrade, pela motivação, e ao TC Claudio Gomes de Mello e ao Maj Julio Cesar Duarte, pela disponibilidade, atenção e, principalmente, pelas relevantes observações realizadas ao longo do curso.

A todos os meus amigos, que contribuiram direta ou indiretamente para realização do meu trabalho.

Por fim, agradeço a todos os professores e funcionários da Seção de Engenharia de Computação (SE/8) do Instituto Militar de Engenharia.

César Augusto Borges de Andrade

”Não há nada permanente, a não ser a mudança”.

Heráclito

SUMÁRIO

LISTA DE ILUSTRAÇÕES 9

LISTA DE TABELAS 11

LISTA DE ABREVIATURAS E SÍMBOLOS 12

1 INTRODUÇÃO 15
1.1 Motivação 15
1.2 Objetivos 17
1.2.1 Principais 17
1.2.2 Secundários 18
1.3 Metodologia 18
1.4 Organização da Dissertação 19

2 ANÁLISE DE MALWARE E APRENDIZADO DE MÁQUINA . . 20

3 APRENDIZADO DE MÁQUINA APLICADO A DETECÇÃO DE MALWARES 26
3.1 Classes de códigos maliciosos 26
3.2 Os Antivírus 29
3.3 Análise de Código malicioso 32
3.3.1 Análise Estática 33
3.3.2 Análise Dinâmica 34
3.4 Análise Automática de Código malicioso 35
3.4.1 Cuckoo Sandbox 37
3.5 Aprendizado de Máquina 39
3.5.1 Algoritmos 40
3.5.2 Naive Bayes 40
3.5.3 SVM 41
3.5.4 Árvores de Decisão 41
3.5.4.1 ID3 43
3.5.4.2 C4.5 43
3.5.4.3 CART 44

3.5.4.4 Random Forest 45
3.5.5 Frameworks de Aprendizado de Máquina 45
3.5.5.1 WEKA 45
3.5.5.2 FAMA 47

4 TÉCNICAS ANTI-ANÁLISE 50
4.1 Empacotamento 50
4.2 Anti-dumping 51
4.3 Anti-debugging 52
4.4 Detecção de Emuladores e Máquinas Virtuais 53
4.5 Sleeping 54
4.6 Anti-disassembly 55

5 ANÁLISE AUTOMÁTICA DE MALWARE UTILIZANDO SANDBOXES E APRENDIZADO DE MÁQUINA 58
5.1 Desdobramento da proposta de sistema automático de análise dinâmica de malware 58
5.1.1 Coleta de Dados 59
5.1.2 Identificação de Comportamento e Geração de Relatórios 59
5.1.3 Análise Comportamental Customizada 61
5.1.3.1 Pré-processamento de Dados 61
5.1.3.2 Aprendizado e Avaliação 63

6 RESULTADOS EXPERIMENTAIS 66
6.1 Experimento 01 67
6.2 Experimento 02 68
6.3 Experimento 03 69
6.4 Experimento 04 70
6.5 Conclusão parcial 71

7 CONSIDERAÇÕES FINAIS 73
7.1 Conclusão 73
7.2 Trabalhos futuros 74

8 REFERÊNCIAS BIBLIOGRÁFICAS 75

LISTA DE ILUSTRAÇÕES

FIG.1.1 Aumento da quantidade de *malwares* novos descobertos durante o período de 2003 a 2012. Os números não são cumulativos, ou seja, o contador é zerado a cada ano. Variantes descobertas derivadas de um tipo de *malware* estão sendo consideradas. Fonte: http://www.av-test.org (AVTEST, 2011). 16

FIG.1.2 Proposta de Sistema Automatizado de Análise Dinâmica de *Malware*. 18

FIG.3.1 Classificação para os programas maliciosos. 27

FIG.3.2 Árvore de classificação dos programas maliciosos segundo Kaspersky Lab. 30

FIG.3.3 Resultado da análise de um *malware* no VirusTotal. 31

FIG.3.4 Análise de código malicioso. 33

FIG.3.5 Tela do *Cuckoo Host* durante o processo de análise do arquivo fotos.exe. 37

FIG.3.6 Arquivos gerados pelo *Cuckoo* ao analisar o *malware* Net-Worm.Win32.Kolabc.buj. 38

FIG.3.7 Parte do relatório gerado pelo *Cuckoo* ao final da analise do *malware* Net-Worm.Win32.Kolabc.buj. 39

FIG.3.8 Treinamento do Classificador de Árvore de Decisão. 42

FIG.3.9 Uma instanciação do Framework Weka para experimentação. 46

FIG.3.10 Seleção do Algoritmo no Weka. 46

FIG.3.11 Classes abstratas do FAMA. 48

FIG.3.12 Instância da Corpus e Avaliador utilizadas no FAMA. 48

FIG.4.1 Criação e execução de programas executáveis empacotados. 51

FIG.4.2 Uso da técnica *anti-debugging* IsDebuggerPresent. 52

FIG.4.3 Uso da técnica *anti-debugging* OutputDebugString. 53

FIG.4.4 Malware usando sleeping para se evadir da análise. 55

FIG.4.5 *Malware* usando o movimento do mouse para se esconder. 55

FIG.4.6 Trecho de código contendo dado no segmento de texto de um binário. 56

FIG.5.1 Fluxo de análise de *malware* (detalhado). 58
FIG.5.2 Arquitetura do ambiente de análise do *Cuckoo Sandbox*. 59
FIG.5.3 Saida do *Script* de submissão automática de *malwares* para análise do *Cuckoo Sandbox*. 60
FIG.5.4 Exemplo de arquivo de vetores de atributos. 63
FIG.5.5 Exemplo de arquivo de vetores de atributos no formato .arff. 64
FIG.5.6 Uma tela do Weka depois de executar o Random Forest. 65
FIG.5.7 Matriz de confusão para um problema com duas classes. 65

FIG.6.1 Resultado do ID3 no FAMA. 72

LISTA DE TABELAS

TAB.2.1 Comparação dos trabalhos relacionados 24

TAB.6.1 Distribuição dos artefatos em cada experimento. 66
TAB.6.2 Distribuição dos artefatos do experimento 01. 67
TAB.6.3 Comparação dos classificadores para detecção de *Worms*. 67
TAB.6.4 Distribuição dos artefatos do experimento 02. 68
TAB.6.5 Comparação dos classificadores para detecção de *Trojans*. 68
TAB.6.6 Distribuição dos artefatos do experimento 03. 69
TAB.6.7 Comparação dos classificadores para detecção de *Backdoors*. 69
TAB.6.8 Distribuição dos artefatos do experimento 04. 70
TAB.6.9 Comparação dos classificadores para detecção de *Malwares*. 70

LISTA DE ABREVIATURAS E SÍMBOLOS

ABREVIATURAS

AM	-	*Aprendizado de Máquina*
API	-	*Aplication Programming Inteface*
BCB	-	*Borland C++ Builder*
CenPRA	-	*Centro de Pesquisas Renato Archer*
FAMA	-	*Framework de Aprendizado de Máquina*
GPL	-	*General Public License*
IDE	-	*Integrated Development Environment*
IME	-	*Instituto Militar de Engenharia*
SVM	-	*Support Vector Machine*
UML	-	*Unified Modeling Language*
WEKA	-	*Waikato Environment for Knowledge Analysis*

RESUMO

A análise de código malicioso (*malware*) permite identificar caracteríticas do comportamento do *software*, ou seja, como atua no sistema operacional, que técnicas de ofuscação são utilizadas, quais fluxos de execução levam ao comportamento principal planejado, uso de operações de rede, operações de *download* de arquivos, captura de informações do usuário ou do sistema, acesso a registros, entre outras atividades, com o objetivo de aprender como o *malware* funciona e criar formas de identificar novos *softwares* maliciosos com comportamento similar assim como formas de defesas.

A análise manual para a geração de assinaturas torna-se inviável, pois demanda muito tempo, se comparado à velocidade de disseminação e criação de novos *malwares*. Sendo assim, a presente dissertação propõe a utilização de técnicas de *sandbox* e aprendizado de máquina para automatizar a identificação de *softwares* nesse contexto.

Esse trabalho, além de apresentar um abordagem diferente e mais rápida para detecção de *malware*, obteve uma taxa de precisão acima de 90%.

ABSTRACT

The analysis of malicious code (malware) identifies caracteríticas of software behavior, in other words, how it acts on the operating system, what kind of obfuscation techniques are used, which streams of execution lead to the main planned behavior, use of network operations, downloading operations of files, capture of user or system's information, access to records, among other activities, in order to learn how the malware works and create ways to identify new malware with similar behavior as forms of defenses.

The manual analysis for the generation of signatures becomes impracticable, since it requires a long time, compared to the speed of dissemination and creation of new malwares. Therefore, this dissertation proposes the use of sandboxes and machine learning techniques to automate the identification and classification of software in this context.

This work, besides presenting a faster and different approach for malware detection, obtained an accuracy rate above 90%.

1 INTRODUÇÃO

Com o crescimento da Internet e dos *softwares* de computadores, a proliferação de *malwares* (*softwares* maliciosos), que são programas desenvolvidos para executar ações danosas em um computador (CERT.BR, 2012), se tornou um problema relevante. Tais programas maliciosos podem, entre outras atividades, roubar informações pessoais e corporativas (GOSTEV, 2012), realizar ataques de negação de serviço (FREILING, 2005) e efetuar transações bancárias ou causar sabotagens industriais (FALLIERE, 2010). O termo *malware* é comumente empregado para se referir a todo e qualquer *software* malicioso. De acordo com o seu comportamento, pode ser classificado como vírus, *worms*, *spywares*, *trojans*, *bot*, entre outros. Um dos mais comuns, os *worms* (SZOR, 2005), são capazes de se multiplicar sem nenhuma intervenção humana, explorando vulnerabilidades nos *softwares* existentes, o que possibilita uma fácil disseminação. O antivírus, principal produto *antimalware*, não consegue acompanhar a criação e a disseminação de tantos *malwares*, pois novas variantes são criadas a todo momento com novas habilidades evasivas, tornando ineficiente as técnicas de análise.

1.1 MOTIVAÇÃO

Atualmente, malwares são criados por uma enorme indústria que se desenvolveu rapidamente e movimenta bilhões de dólares. Um exemplo do alto nível de investimento em projetos de criação de *malwares* é o Stuxnet. Segundo (CHEN, 2011), o *worm* tem um alto grau de complexidade, que requer diferentes habilidades para criá-lo.

Além da sofisticação, uma outra preocupação diz respeito a grande quantidade de novos *malwares*. O número de amostras únicas de *malwares* registrado pela (AVTEST, 2011) tem aumentado muito nos últimos 10 anos, passando da marca dos 30 milhões em dezembro de 2012 (FIG. 1.1).

A indústria de antivírus reagiu à evolução dessa situação através da distribuição mais frequente de atualizações para assinaturas de vírus. Alguns fornecedores mudaram de atualizações semanais para diárias, ou mesmo, atualizações a cada meia hora.

A análise manual para a geração de assinaturas torna-se, portanto, inviável, pois demanda muito tempo, se comparado à velocidade de disseminação e criação de novos

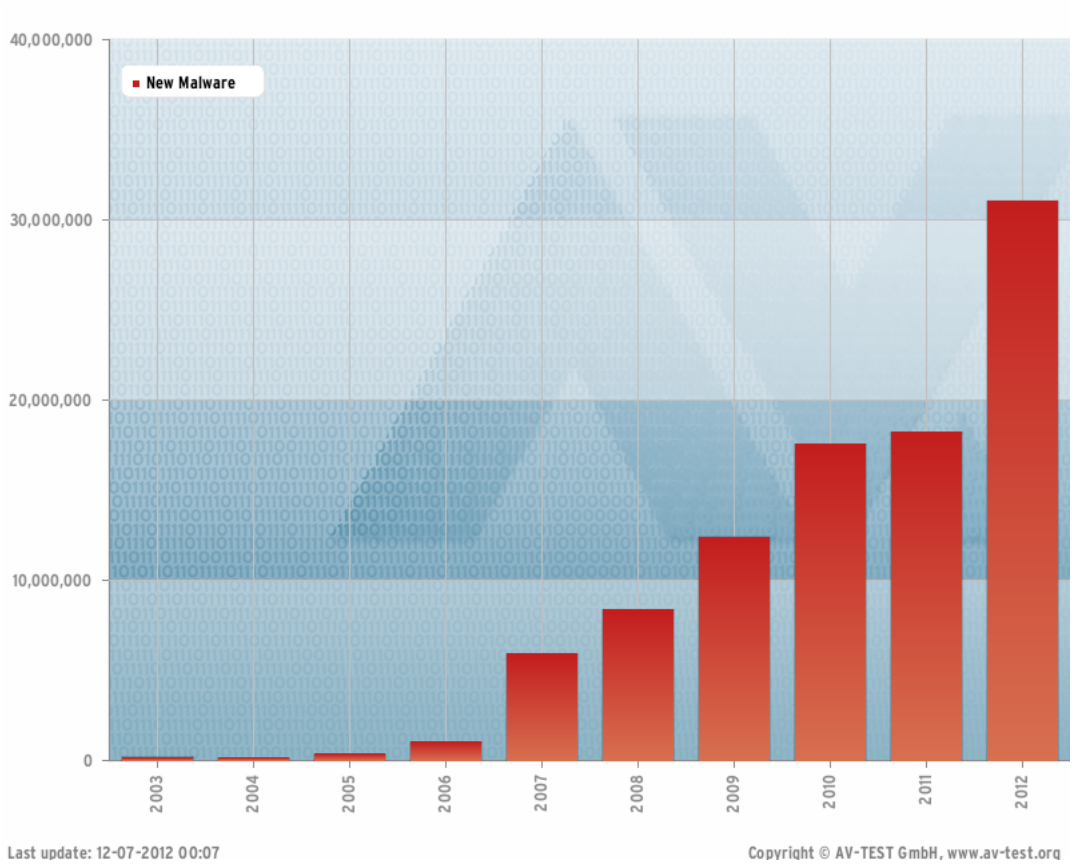

FIG. 1.1: Aumento da quantidade de *malwares* novos descobertos durante o período de 2003 a 2012. Os números não são cumulativos, ou seja, o contador é zerado a cada ano. Variantes descobertas derivadas de um tipo de *malware* estão sendo consideradas. Fonte: http://www.av-test.org (AVTEST, 2011).

malwares.

Com o objetivo de prover a segurança de um ambiente computacional, há a necessidade de detectá-los de maneira eficiente. Neste cenário, a análise automática se mostra uma opção mais eficiente para o processo. Essa análise é obtida através de mecanismos de restauração automática de ambientes de testes como máquinas virtuais (GREAMO, 2011).

Como visto no modelo proposto por (ZOLKIPLI, 2010), um dos grandes problemas da análise automática é que a interpretação dos grandes relatórios gerados por *sandboxes* (ambientes restritos e controlados para execução de artefatos, geralmente, *softwares* suspeitos) é deixada a cargo do usuário, ou seja, não se pode realmente dizer que o sistema realizou uma análise, mas sim, confeccionou um relatório de sua execução, com o registro das atividades efetuadas em um determinado período.

Alguns países, vendo a motivação política por trás de alguns ataques de *malwares*,

já estão desenvolvendo seus programas de guerra cibernética. Os EUA estabeleceram um *Cyber Command* (USCYBERCOM) em *Fort Meade*, *Maryland*, para defender as redes militares americanas. Outros países, incluindo Reino Unido, Irã, Turquia, Coréia do Norte e Sul, China, e a Federação Russa, estão buscando as capacidades necessárias a guerra cibernética. Applegate (APPLEGATE, 2011) nos chama a atenção principalmente para a República Popular da China e a Federação Russa como dois exemplos de nações que executam esses tipos de ataques, e da ambiguidade em torno deles. Rússia e China estão agressivamente desenvolvendo programas de guerra cibernética, e a China tem desenvolvido um grande corpo de doutrina e publicações profissionais apoiando este novo conceito.

No Brasil, o Setor Cibernético recebeu destaque na última edição da Estratégia Nacional de Defesa (PRESIDÊNCIA, 2008) e, em agosto de 2010, foram aprovadas as portarias 666 e 667 (EXÉRCITO, 2010), do Comandante do Exército, criando o Centro de Defesa Cibernética do Exército (CDCiber) e ativando o Núcleo do Centro de Defesa Cibernética do Exército (Nu CDCiber), respectivamente.

1.2 OBJETIVOS

Este trabalho propõe o desenvolvimento de técnicas de análise de *malware* com a utilização de *sandboxes* e aprendizado de máquina, para automatizar a identificação de códigos maliciosos. O esquema geral da proposta é apresentado na FIG. 1.2, onde o módulo da Análise Comportamental Customizada deixará de sofrer a intervenção humana e passará a ser realizada por técnicas de Aprendizado de Máquina.

Durante a pesquisa, foram traçados alguns objetivos:

1.2.1 PRINCIPAIS

- instalação e configuração de um *sandbox*, que é um ambiente seguro e controlado para execução dos artefatos;
- criação de um mecanismo que permita a submissão automática para a análise de milhares de artefatos; e
- armazenamento dos relatórios produzidos pelo *sandbox*.

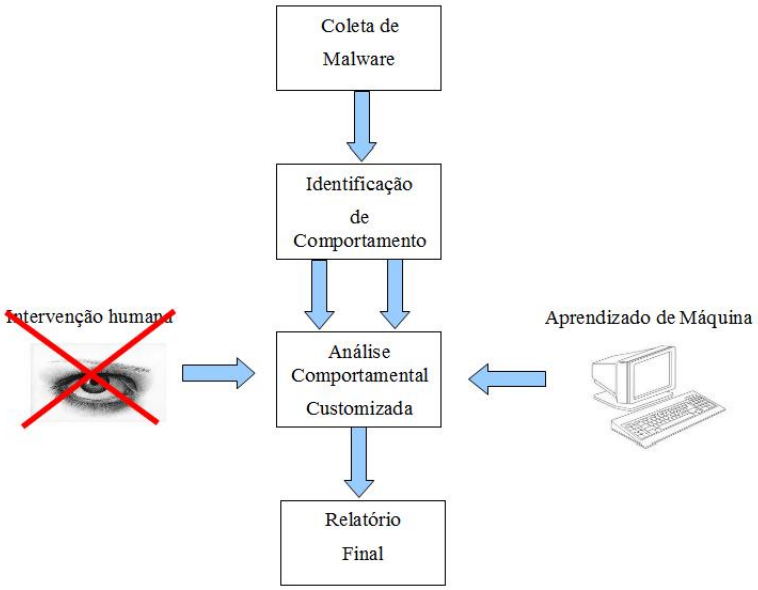

FIG. 1.2: Proposta de Sistema Automatizado de Análise Dinâmica de *Malware*.

1.2.2 SECUNDÁRIOS

- tratamento dos relatórios produzidos pelo *sandbox*;
  - alimentação dos algoritmos de AM com os relatórios sanitizados;
  - selecionar o algoritmo que apresentar melhor desempenho;
  - implementar o algoritmo selecionado na etapa anterior; e
  - analisar o desempenho do algoritmo

1.3 METODOLOGIA

Para que esses objetivos fossem alcançados, foram dados os seguintes passos:

1- instalação e configuração de um *sandbox Open Source*.

2- implementação de um script para automatizar a submissão de artefatos.

3 - cada relatório produto da análise foi armazenado em subdiretório junto com o artefato analisado.

4 -implementação de um script que transforma os relatórios em um formato adequado para alimentar os algoritmos de AM.

5 - utilização um *framework* de AM para selecionar o algoritmo de melhor desempenho.

6 - implementação do algoritmo selecionado na etapa anterior.

7 - análise dos resultados.

1.4 ORGANIZAÇÃO DA DISSERTAÇÃO

Esta dissertação está organizada da seguinte forma: após a introdução, são discutidos no capítulo 2 os trabalhos relacionados. No capítulo 3, são abordados conceitos referentes à análise de código malicioso e aprendizado de máquina. No capítulo 4, são apresentadas algumas técnicas de anti-análise utilizadas por *malwares*. No capítulo 5, são apresentadas as características do experimento realizado, bem como a metodologia utilizada. No capítulo 6, é apresentada a avaliação dos resultados obtidos. No capítulo 7, são realizadas as considerações finais e apresentadas sugestões de trabalhos futuros.

2 ANÁLISE DE MALWARE E APRENDIZADO DE MÁQUINA

O constante surgimento de novas variantes, o emprego de técnicas complexas de ofuscação e evasão e, muitas vezes, a disponibilidade de grandes recursos para o seu desenvolvimento, torna o *malware* o grande vilão cibernético dos nossos dias. Com toda essa relevância, não é surpreendente que uma quantidade significativa de pesquisadores tenha concentrado esforços no desenvolvimento de técnicas para coletar, estudar e mitigar códigos maliciosos. Por exemplo, há estudos que analisam o funcionamento de uma *bot* (um programa que dispõe de mecanismos de comunicação com o invasor que permitem que o sistema da vítima seja controlado remotamente) para depois infiltrar-se na *botnet* e desativá-la (FREILING, 2005), também há estudos focados no desenvolvimento de diversos mecanismos que visam obter o código não ofuscado do *malware*, permitindo que a análise estática seja efetuada (SHARIF, 2008), (KANG, 2007) e (MARTIGNONI, 2007).

Abordagens baseadas em aprendizado de máquina para detecção de códigos maliciosos utilizam algoritmos para treinamento dos dados no intuito de detectar *malwares*. Alguns exemplos de aplicações para detecção de *malwares* utilizando aprendizado de máquina são apresentados a seguir.

(SCHULTZ, 2001) foi o primeiro trabalho a introduzir o conceito de aplicação de algoritmos de aprendizado automático para a detecção de *malware*, com base em seus respectivos códigos binários, aplicando diferentes classificadores como RIPPER e Naive Bayes (JOHN, 1995), e considerando três tipos de conjuntos de características: (i) cabeçalhos de programa, (ii) cadeias de caracteres e (iii) sequências de *bytes*. Posteriormente, (KOLTER, 2004) melhorou os resultados de Schultz, aplicando n-gramas, ou seja, sobrepondo sequências de *bytes*. Nesse caso, os resultados experimentais obtidos com Árvore de Decisão foram superiores aos demais algoritmos de aprendizado de máquina investigados nos experimentos.

(WANG, 2003) propõe um método de detecção de vírus, até então desconhecido, utilizando técnicas de aprendizado de máquina como Árvore de Decisão e *Naive Bayes*. O método utiliza uma base de dados contendo 3.265 códigos maliciosos e 1.001 benignos. Todo o processo de extração de características foi realizado automaticamente, a partir de uma abordagem estática. Por fim, os vetores foram utilizados para treinar os classifi-

cadores e detectar os possíveis vírus. Resultados experimentais mostraram que a precisão do modelo utilizando Árvore de Decisão foi superior a *Naive Bayes*, obtendo precisão de 91,4% e 77,1%, respectivamente.

Em (KOLTER, 2006), foi desenvolvido MECS (*Malicious Executable Classification System*) para detectar automaticamente executáveis maliciosos, sem preprocessamento ou remoção de qualquer ofuscação. O sistema utiliza um conjunto de 1.971 executáveis benignos e 1.651 executáveis maliciosos. Embora todos os códigos sejam executados no sistema operacional *Microsoft Windows*, vale ressaltar que o método não se restringe unicamente a este sistema operacional. As sequências de *bytes* dos executáveis foram convertidas em n-gramas e, então, indexadas no vetor de características, que por sua vez foi repassado como parâmetro de entrada para treinamento e teste de diversos classificadores como: *k-Nearest Neighbours* (MITCHELL, 1997), *Naive Bayes* (JOHN, 1995), *Support Vector Machines* (CHANG, 2011) e Árvore de Decisão (QUINLAN, 1986). Neste domínio, os autores atentaram para o problema de falsos positivos, o que ocasiona custos desiguais de classificação. Em termos de desempenho, a Árvore de Decisão superou os diversos outros métodos, obtendo precisão de 99,58%.

Em (YE, 2007), os autores apresentam o IMDS (*Intelligent Malware Detection System*), ou seja, um sistema inteligente para detecção de malwares, mais precisamente, vírus polimórficos. O IMDS é baseado em técnicas de mineração de dados. O sistema realiza análise de sequências de execução de diversas APIs do *Windows*, que refletem o comportamento dos códigos maliciosos. Em seguida, a detecção de *malware* é realizada diretamente no PE (*Windows Portable Executable*), que é o formato padrão dos executavéis do *Windows*, em três etapas: i) construção das sequências de execução de API através do desenvolvimento de um analisador PE; ii) extração de regras usando técnicas de mineração de dados e; iii) classificação baseada nas regras de associação geradas na segunda etapa. O sistema utiliza uma base de dados contendo 29.580 executáveis, sendo 12.214 códigos benignos e 17.366 maliciosos. O sistema IMDS foi comparado com diversos antivírus populares como Norton AntiVirus e McAfee, sendo que os resultados experimentais foram superiores à eficiência dos demais antivírus. Para isso, foram utilizadas técnicas de aprendizado de máquina como *Naive Bayes*, SVM e DT. Este é um exemplo que mostra o resultado de aplicações de técnicas de aprendizado de máquina em segurança de redes, pois a abordagem introduzida pelo IMDS resultou na incorporação do sistema à ferramenta de verificação do Kingsoft Antivirus.

(RIECK, 2008) apresentou a proposta de uma abordagem de classificação automática de comportamento de um malware, tendo como base os seguintes itens: i) a incorporação de rótulos atribuídos pelo software antivírus para definir classes para a construção de modelos supervisionados; (ii) o uso de recursos específicos, como cadeia de caracteres, que descrevem padrões de comportamento de *malware*; iii) a construção automática de modelos discriminativos utilizando algoritmos de aprendizado como SVM e; iv) identificação das características dos modelos explicativos que mais influenciaram o aprendizado, produzindo um ranking de padrões de comportamento de acordo com seus pesos. Na prática o método funciona da seguinte forma: coleta-se um número de amostras de *malwares* e analisa-se seu comportamento usando um ambiente *sandbox* (ex: CWSandbox), identificam-se *malware* típicos a serem classificados, executando um *software* antivírus padrão (ex:Avira), e constrói-se um classificador baseado no comportamento do *malware* aprendendo as classes simples dos modelos. A precisão obtida pelo método foi equivalente a 70%.

(RIECK, 2011) propõe um *framework* para análise automática de *malware* utilizando técnicas de aprendizado de máquina. O *framework* permite agrupar automaticamente as novas classes de *malware* com comportamento semelhante (agrupamento) e associar um novo *malware* (desconhecido) a uma destas classes descobertas (classificação). Os artefatos foram executados e monitorados utilizando-se o CWSandbox. Os algoritimos de aprendizado de máquina são alimentados por uma representação chamada *Malware Instruction Set* (MIST) proposta por (TRINIUS, 2010), que é uma forma mais otimizada dos relatórios gerados pelo *sandbox*.

Um outro trabalho que tem a mesma preocupação do (TRINIUS, 2010), ou seja, na representação dos dados dos enormes relatórios gerados por *sandboxes* é o (LI, 2011). Nele é apresentado um método otimizado de *sandbox* usado na classificação de *malwares* com base no comportamento. É utilizado o CSS (*Crystal Security Sandbox*) para monitorar a execução de arquivos (PE) e gerar relatórios intermediários e sanitizados. Embora o resultado intermediário seja suficiente para descrever o comportamento de *malware*, é ainda um pouco longo, muito redundante, e grande demais para ser analisado com eficiência por um ser humano. Portanto, esse relatório é comprimido e informações desnecessárias são removidas resultando em outro relatório 90% menor. Além da redução do tamanho dos dados, os autores tem a preocupação de também manter a taxa de precisão acima de 93% e a taxa de erro abaixo de 7%. O sistema usa o ClamAV-Antivirus para rotular os *malwares* e SVM para treinar o classificador.

Através de uma abordagem estática, (RAMAN, 2012) identifica sete atributos de maior relevância nos arquivos de formato PE. Com o objetivo de criar modelos para classificar malwares, esses poucos atributos podem ser usados como parâmetros de entrada nos algoritmos de aprendizado de máquina. No experimento, foram analisados uma grande quantidade de *malwares* e não *malwares*. Foram utilizados 6 algoritmos de classificação e o J48 obteve o melhor resultado. Esse estudo apresentou resultados comparáveis a outras pesquisas existentes na área, que utilizam uma maior quantidade de atributos (SIDDIQUI, 2009), (SHAFIQ, 2009) e (KHAN, 2011).

(SINGHAL, 2012) propõe um novo e sofisticado motor antivírus que não só pode varrer arquivos, mas também construir conhecimento e classificar arquivos como possíveis *malwares*. Isto é feito através da extração de chamadas do sistema (API) e o uso de algoritmos de aprendizado de máquina para classificar e ordenar os arquivos em uma escala de risco de segurança. Dos 5000 executáveis analizados (combinação de malignos e benignos), 4.470 foram classificados corretamente. A TAB. 2.1 mostra uma comparação dos trabalhos descritos anteriormente.

Muito avanço tem sido feito no estudo da análise estática, mas essa abordagem continua sendo ineficiente frente ao constante emprego das técnicas de ofuscação como empacotamento, criptografia, polimorfismo e metamorfismo (CHRISTODORESCU, 2003), (MOSER, 2007), (KRUEGEL, 2005) e (PREDA, 2007). Ao contrário da análise estática, a análise dinâmica de binários permite monitorar o comportamento do *malware* durante a sua execução, dificultando o ocultamento de comportamento suspeito e oferecendo indicativos de atividades maliciosas. Assim, uma substancial quantidade de pesquisadores tem focado esforços no desenvolvimento de ferramentas para coleta e monitoramento de *malware* (POUGET, 2005), (CAVALCA, 2010), (BAECHER, 2006), (BAYER, 2006a), (BAYER, 2006b) e (WILLEMS, 2007).

Tendo em mente as limitações da abordagem estática e buscando traçar de forma automática o perfil dos códigos maliciosos, a presente proposta trabalha com a abordagem dinâmica. E diferente dos trabalhos mencionados anteriormente, que utilizam a abordagem dinâmica para classificação de uma classe de *malware* em subclasses e a abordagem estática para detecção (classificação binária: malignos e benignos), propomos fazer detecção utilizando análise dinâmica. Outro diferencial dos trabalhos citados, que trabalharam com *sandboxes* proprietários ou serviços *on-line* oferecidos gratuitamente por empresas de segurança, é a busca da independência de terceiros instalando e personali-

TAB. 2.1: Comparação dos trabalhos relacionados

Publicação	Abordagem	Técnica de AM	Base	Precisão
(SCHULTZ, 2001)	Estática	RIPPER e *Naive Bayes*	3.265 malignos e 1.001 benignos	71,05% RIPPER e 97,76% *Naive Bayes*
(WANG, 2003)	Estática	Árvore de decisão e *Naive Bayes*	3.265 malignos e 1.001 benignos	91,4% AD e 77,1% *Naive Bayes*
(KOLTER, 2006)	Estática	Árvore de decisão, KNN, SVM e *Naive Bayes*	1.651 malignos e 1.971 benignos	99,58% AD, 98,99% KNN, 99,03% SVM e 98,87% *Naive Bayes*
(YE, 2007)	Estática	Árvore de decisão, SVM e *Naive Bayes*	17.366 malignos e 12.214 benignos	91,49% AD, 90,54% SVM e 83,86% *Naive Bayes*
(RIECK, 2008)	Dinâmica	SVM	10.072 malignos	70,00%
(RIECK, 2011)	Dinâmica	SVM	3.133 malignos	96,00%
(LI, 2011)	Dinâmica	SVM	600 malignos	93,00%
(RAMAN, 2012)	Estática	Árvore de decisão, PART, Ridor e *Random Forest*	100.000 malignos e 16.000 benignos	98,56% AD, 98,22% PART, 97,92% Ridor e 98,22 *Random Forest*
(SINGHAL, 2012)	Estática	Árvore de decisão, *Naive Bayes* e *Random Forest*	5.000 executáveis	90% AD, 95% *Naive Bayes*, 97% *Random Forest* e 99.55% proposta

zando uma ferramenta que tem seu código disponível para estudo, o *Cuckoo*. Para as tarefas de aprendizado de máquina, foram utilizados dois *frameworks*: o (WEKA, 1993) e o (FAMA, 2011). O FAMA, vem sendo desenvolvido no IME desde 2011. Durante os trabalhos, foram implementados os algoritmos ID3 e *Random Forest* no *Framework* FAMA.

3 APRENDIZADO DE MÁQUINA APLICADO A DETECÇÃO DE MALWARES

3.1 CLASSES DE CÓDIGOS MALICIOSOS

Códigos maliciosos podem ser classificados de uma maneira geral de acordo com características específicas de seu comportamento. Embora atualmente seja difícil classificar um exemplar de *malware* em uma única classe, devido à evolução destes códigos e à facilidade de se adicionar novas funcionalidades, a taxonomia apresentada a seguir ainda é utilizada para se referir a certos tipos de malware e também nos identificadores atribuídos por mecanismos antivírus.

As classes e descrições abordadas a seguir baseiam-se nas definições de (SZOR, 2005).

Vírus. Segundo Fred Cohen (COHEN, 1987), considerado o pai dos vírus de computador, um vírus é um programa que é capaz de infectar outros programas pela modificação destes de forma a incluir uma cópia possivelmente evoluída de si próprio. Os vírus costumam infectar seções de um arquivo hospedeiro, propagando-se através da distribuição deste arquivo. Comumente, os vírus necessitam ser acionados e propagados por uma entidade externa (usuário).

Worm. Os *worms*, por sua vez, propagam-se pela rede e em geral não necessitam de ativação por parte do usuário. Uma outra característica é a independência de outro programa para disseminação e ataque. Alguns worms podem carregar dentro de si um outro tipo de *malware*, que é descarregado na vítima após o sucesso da propagação e do ataque.

Trojan. Cavalos-de-tróia são tipos comuns de *malware* cujo modo de infecção usa a curiosidade do usuário para que este o execute e comprometa o sistema. Este tipo de código malicioso também pode ser encontrado em versões modificadas de aplicações do sistema operacional, substituídas por indivíduos maliciosos. Estas versões apresentam as mesmas funcionalidades da aplicação legítima, porém também contém funcionalidades adicionais com a finalidade de ocultar as ações malignas.

Backdoor. Tradicionalmente, *backdoors* permitem a um atacante a revisitação de uma máquina comprometida por abrirem portas que permitem a conexão remota. Um

outro tipo de *backdoor* envolve erros na implementação de uma aplicação ou sistema que o tornam vulnerável e podem levar à execução de código arbitrário na máquina da vítima.

Downloader. Um programa malicioso que conecta-se à rede para obter e instalar um conjunto de outros programas maliciosos ou ferramentas que levem ao domínio da máquina comprometida. Para evitar dispositivos de segurança instalados na vítima, é comum que *downloaders* venham anexos à mensagens de correio eletrônico e, a partir de sua execução, obtenham conteúdo malicioso de uma fonte externa. (ex: site).

Dropper. Possui características similares às dos *downloaders*, com a diferença que um *dropper* é considerado um instalador, uma vez que contém o código malicioso compilado dentro de si.

Rootkit. É um tipo especial de *malware*, pois consiste de um conjunto de ferramentas para possibilitar a operação em nível mais privilegiado. Seu objetivo é permanecer residindo no sistema comprometido sem ser detectado e pode conter *exploits*, *backdoors* e versões *trojans* de aplicações do sistema. Os *rootkits* modernos atacam o *kernel* do sistema operacional, modificando-o para que executem as ações maliciosas de modo camuflado. Este tipo de *rootkit* pode inclusive interferir no funcionamento de mecanismos de segurança

(SZOR, 2005) propõe a classificação que é apresentada para os diferentes códigos maliciosos (FIG. 3.1).

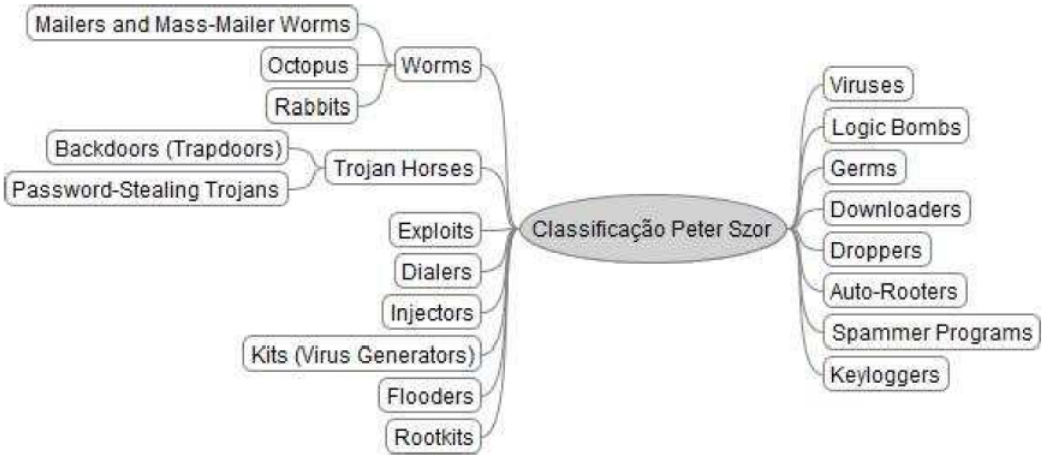

FIG. 3.1: Classificação para os programas maliciosos.

O (CERT.BR, 2012) adiciona a descrição de mais outras classes de malwares:

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

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

Segundo o (CERT.BR, 2012), existem diferentes tipos de *trojans*, classificados de acordo com as ações maliciosas que costumam executar ao infectar um computador. Alguns destes 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 ou Bancos: coleta dados bancários do usuário, através 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.

A classificação utilizada como referência nesse trabalho é a proposta na *Securelist* da (KASPERSKY, 2012) (FIG. 3.2), em virtude da necessidade de identificar programas maliciosos utilizados na experimentação e também da similaridade da classificação descrita pelo (CERT.BR, 2012).

3.2 OS ANTIVÍRUS

A medida que as ameaças de softwares maliciosos mudaram, os produtos *antimalware* tiveram que evoluir também. O mais popular ainda é o antivírus, que é basicamente um programa que varre arquivos e/ou monitora ações pré-definidas em busca de indícios de atividades maliciosas. Em geral, os antivírus trabalham de duas maneiras para identificar *malwares*: assinaturas e/ou heurísticas. Na detecção por assinatura, um arquivo executável é dividido em pequenas porções (blocos) de código, as quais são comparadas com a base de assinaturas do antivírus. Assim, se um ou mais blocos do arquivo analisado estão presentes na base de assinaturas, a identificação relacionada é atribuída ao referido arquivo. Detecção baseada em assinaturas simples provou ter desvantagens ao ter que lidar com milhares de novas amostras de malwares diariamente. Enquanto novas assinaturas são criadas, o sistema do cliente permanece desprotegido contra essa ameaça particular. Por isso a detecção heurística foi introduzida. Esta técnica também executa varredura estática de *malware*, mas não depende de assinaturas: utiliza técnicas e padrões mais genéricos. No entanto, o sucesso desta técnica nem sempre é garantida, por isso, novamente, algumas abordagens mais sofisticadas são necessárias para detectar os *malwares*. O grande problema dos antivírus é o surgimento frequente e crescente de variantes de malware previamente identificados, cujas ações modificadas visam evadir a detecção. Essas variantes precisam ser tratadas muitas vezes individualmente (e manualmente), no caso

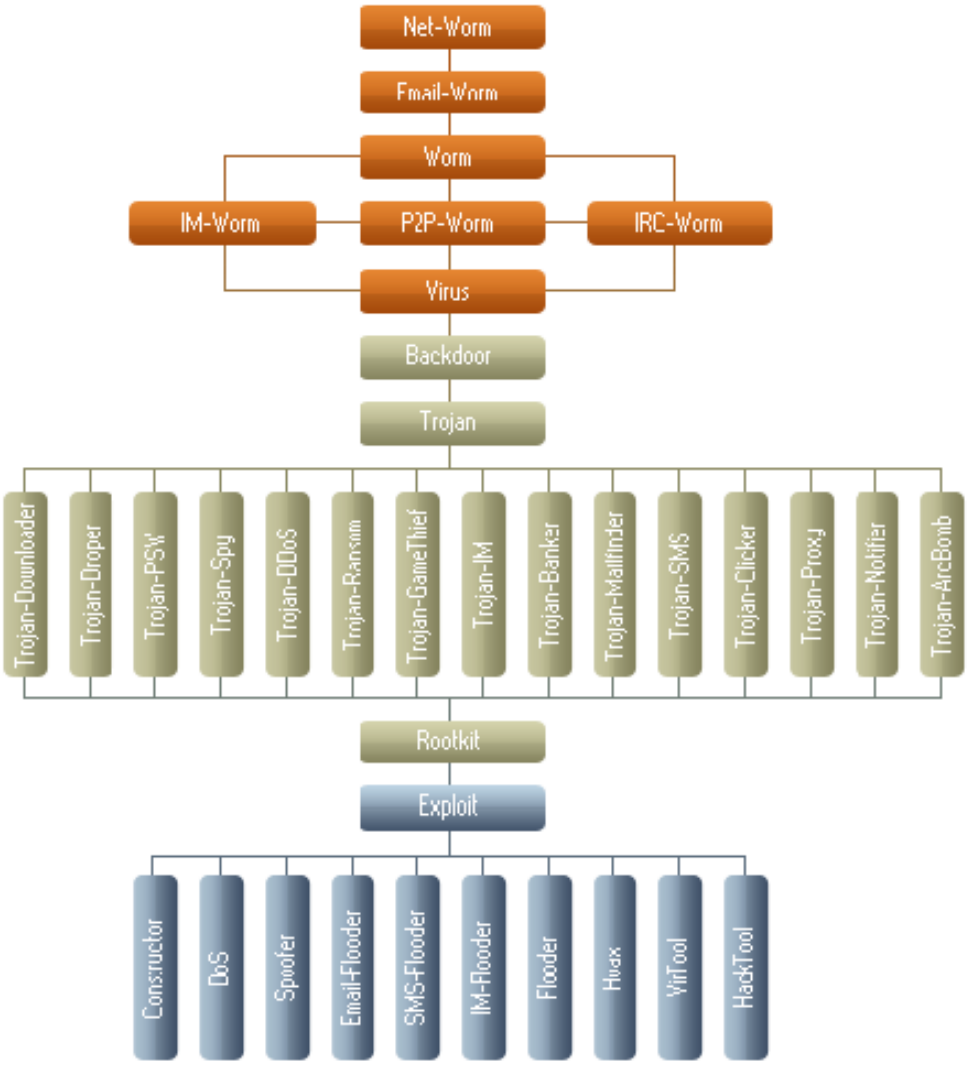

FIG. 3.2: Árvore de classificação dos programas maliciosos segundo Kaspersky Lab.

da criação de uma assinatura para um antivírus. Temos que levar em conta que muitos *malwares* possuem mecanismos próprios de defesa cujas ações variam entre desabilitar as proteções existentes no sistema operacional alvo (*firewall*, antivírus, *plugins* de segurança), verificar se o exemplar está sob análise (o que pode causar uma modificação em seu comportamento em razão disto), e disfarçar-se de programas legítimos do sistema, inclusive de falsos antivírus. Portanto, evitar que o mecanismo antivírus seja desativado, identificar programas maliciosos com eficácia e rapidez e ainda ser transparente aos usuários têm se tornado cada vez mais difícil. Adicione a esse quadro a facilidade de criação de variantes de *malwares* já existentes, popularização das redes sociais, proliferação de celulares e *tablets*, e o estabelecimento da *cloud computing*. Como não existe uma base única e

padronizada de assinaturas para a classificação dos *malwares* identificados, cada empresa desenvolve a sua, o que influencia diretamente na eficiência dos antivírus. Isso faz com que o processo de resposta a incidentes de segurança envolvendo código malicioso seja prejudicado, tornando-o ineficiente em determinados casos. Uma iniciativa interessante e consolidada na comunidade de segurança da informação é o (VIRUSTOTAL, 2011). O VirusTotal é uma ferramenta que atualmente permite a análise de um arquivo binário em 43 diferentes sistemas de antivírus. Além disso, o sistema informa a assinatura do *malware* analisado, caso o mesmo tenha sido detectado como malicioso. O resultado do processamento da ferramenta é apresentado na FIG. 3.3.

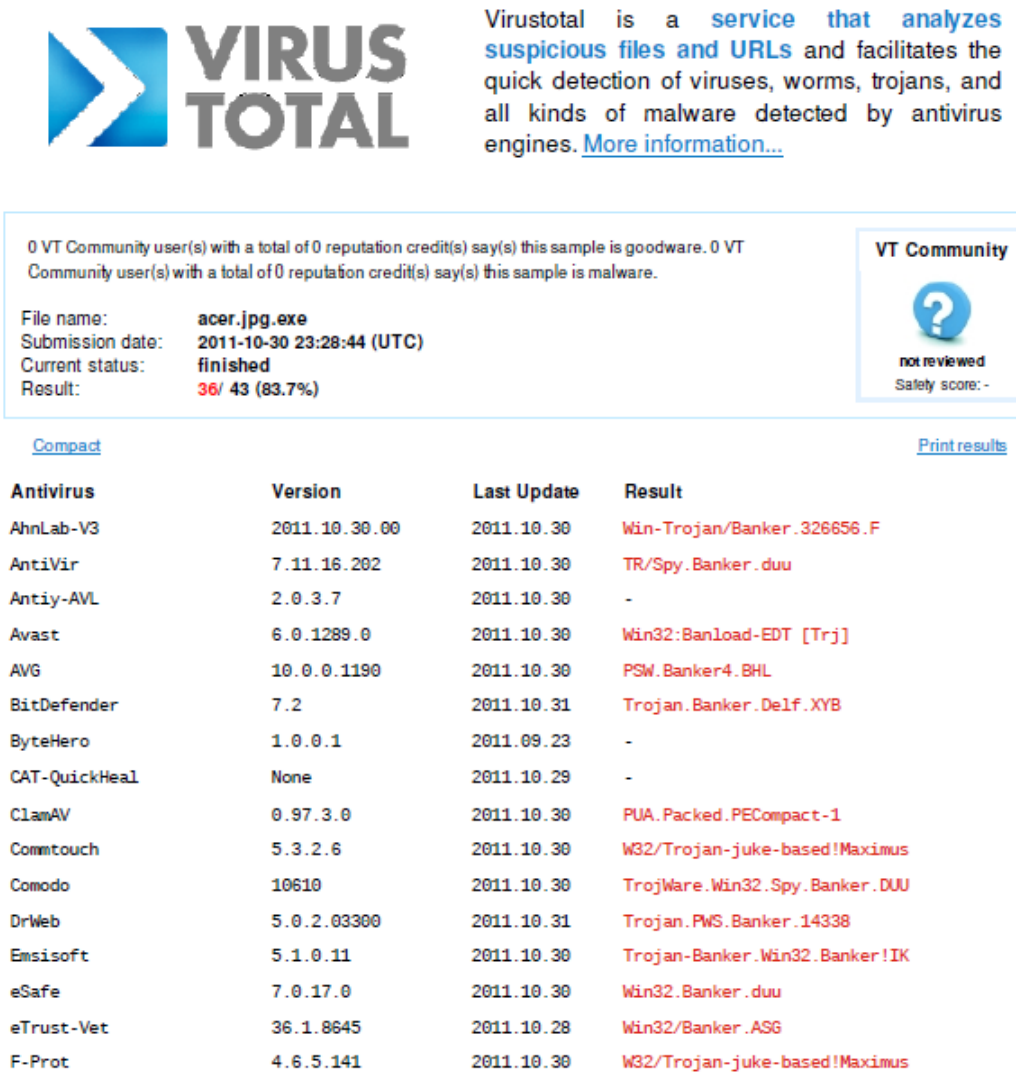

FIG. 3.3: Resultado da análise de um *malware* no VirusTotal.

É possível observar na parte superior que o sistema analisou o binário acer.jpg.exe em

43 diferentes antivírus, e 36 deles detectaram uma assinatura para o *malware*, ou seja uma taxa de detecção de 83,7%.

Uma outra iniciativa mais recente e nacional é o (VIRUSLAB, 2012). O Viruslab é um portal disponibilizado de forma gratuita a toda comunidade de forma que possa ser feita a análise de arquivos suspeitos avaliando se eles são maliciosos ou não através da verificação de diversos antivírus. O usuário dessa forma pode analisar qualquer arquivo e em qualquer lugar que esteja sem instalar nada em seu computador, bastando para isso apenas uma conexão de Internet. Atualmente a ferramenta permite a análise de um arquivo binário em 11 diferentes sistemas de antivírus.

Para alguns casos mais relevantes, as fábricas de antivírus disponibilizam aplicações ou rotinas automatizadas para remoção, as quais podem ser inócuas na presença de variantes. Entretanto, a obtenção do procedimento bem sucedido de remoção (automático ou manual) correto é completamente dependente da identificação provida pelo mecanismo antivírus. (CHRISTODORESCU, 2004) e (YOU, 2010) mostraram que o antivírus, a principal ferramenta para combater *malwares*, é ineficiente quando a ameaça é desconhecida ou quando utiliza meios evasivos.

3.3 ANÁLISE DE CÓDIGO MALICIOSO

A análise de código malicioso tem por objetivo alcançar o entendimento profundo do funcionamento de um *malware* - como atua no sistema operacional, que tipo de técnicas de ofuscação são utilizadas, quais fluxos de execução levam ao comportamento principal planejado, se há operações de rede, *download* de outros arquivos, captura de informações do usuário ou do sistema, entre outras atividades. Como pode ser visto na FIG. 3.4, a análise de *malware* divide-se em análise estática e dinâmica, sendo que no primeiro caso tenta-se extrair características de seu código sem executá-lo, através de análise de *strings*, *disassembler* (ex: ferramenta IDAPro) e engenharia reversa, por exemplo. Já na análise dinâmica, o *malware* é monitorado durante sua execução, por meio de emuladores, *debuggers*, máquinas virtuais, *sandboxes*, ferramentas para monitoração de processos, registros e arquivos e traços de chamadas de sistema.

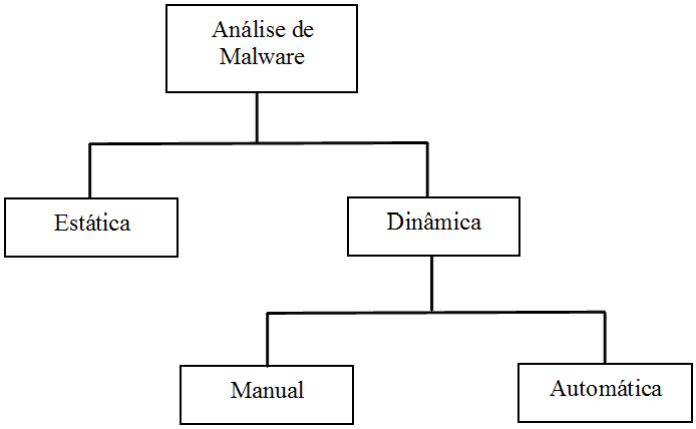

FIG. 3.4: Análise de código malicioso.

3.3.1 ANÁLISE ESTÁTICA

A análise estática é uma técnica utilizada para coletar informações gerais sobre o programa e para identificar a existência de código malicioso sem a necessidade de executá-lo. A análise envolve o uso de um *disassembler* que faz a conversão do código binário para código *Assembly*. A análise estática exige um entendimento de *Assembly* e programação, pois o fluxo da execução é somente baseado no código do programa. Ferramentas de análise de fluxo de controle coletam informações para uma melhor compreensão do fluxo de execução do programa. Como o fluxo pode ser dependente dos valores contidos em posições de memória, a análise do fluxo de dados prediz o conjunto de valores que a memória pode vir a assumir durante a execução do programa. Outra ferramenta que se enquadra em análise estática são os decompiladores que tentam reverter o processo de compilação para produção de um código de alto nível (ex: linguagem C). Contudo, na maioria das arquiteturas, a recuperação do código de alto nível não é realmente possível, pois existem elementos significativos que são removidos durante a compilação, uma vez que o objetivo deste é otimização e não a legibilidade do código. Dentre as técnicas utilizadas para a obtenção de informações gerais estão a geração de *hashes* criptográficos (ex: MD5 e SHA-1) que identificam o arquivo de forma única, a identificação das funções importadas e exportadas, a identificação de código ofuscado e a obtenção de cadeias de caracteres (*strings*) que possam ser lidas por uma pessoa, como mensagens de erro, URLs e endereços de correio eletrônico.

Para identificar código malicioso, são usadas, de forma geral, duas abordagens: a verificação de padrões no arquivo binário e a análise do código *assembly* gerado a partir

do código de máquina do *malware*. No primeiro caso, são geradas sequências de *bytes*, chamadas de assinaturas, que identificam um trecho de código frequentemente encontrado em programas maliciosos e é verificado se o programa possui esta sequência. Os códigos maliciosos além de suas próprias assinaturas podem carregar assinaturas de ferramentas utilizadas durante seu desenvolvimento como compiladores, *packers*, cifradores e etc. Um exemplo de um banco de dados de assinaturas dessas ferramentas é o banco contido na ferramenta PEId, onde contém mais de 600 assinaturas. Já no segundo caso, são empregadas técnicas de análise mais complexas que buscam padrões de comportamentos maliciosos. Atualmente, o maior inimigo da análise estática é o uso crescente dos packers, que são programas feitos para ofuscar o código malicioso e dificultar sua detecção e análise. Segundo (O'KANE, 2011), uma parte substancial do arsenal de malwares escritos utiliza *packers*. Para combater a evolução destes, foram desenvolvidos diversos mecanismos (SHARIF, 2008), (KANG, 2007) e (MARTIGNONI, 2007) que visam obter o código original (desempacotado) do *malware*, permitindo que a análise estática seja efetuada. Apesar de vários esforços no sentido de criar um desempacotador universal (MARTIGNONI, 2007) e (ROYAL, 2006), o mesmo não existe para abordagem estática. De forma geral, a análise estática é pouco eficiente para programas que utilizam técnicas de ofuscação ou polimorfismo (LINN, 2003), como é o caso do *malware* Conficker (LEDER, 2009).

3.3.2 ANÁLISE DINÂMICA

A análise dinâmica de programas é uma técnica em que a coleta de informações é feita em tempo de execução diante de uma determina entrada. Como pode ser visto na (FIG. 3.4), a análise dinâmica é dividida em análise manual e automática.

Na análise manual, é possível observar como a entrada interage com o fluxo de execução e os dados do programa. Essa análise utiliza um depurador (*debugger*) que permite a observação de um programa durante sua execução ou um ambiente controlado como uma máquina virtual (VMWare, VirtualBox e outros) com programas de monitoramento. Um depurador possui tipicamente duas características básicas: habilidade de ajustar pontos de parada (*breakpoints*) no programa e capacidade de verificar o estado atual do programa (registradores, memória e conteúdo da pilha).

Na análise automática, o código é executado num *sandbox*, que no contexto de nosso trabalho, é um ambiente restrito e controlado, que permite a execução de um código malicioso de forma a causar danos mínimos aos sistemas externos por meio da combinação

de filtragem e bloqueio de tráfego de rede e da execução temporizada do *malware*. Em geral, o *malware* é executado por quatro ou cinco minutos e, durante este tempo, são monitoradas de forma automática as ações pertinentes tanto ao *malware* quanto aos processos derivados dele. Após o período de monitoramentp, um relatório de atividades é gerado para análise.

Os *sandboxes* geralmente utilizam duas tecnologias para sua implementação: emulação e virtualização. Na emulação, o código é executado em um emulador que é um software que reproduz as funções de um determinado ambiente, a fim de permitir a execução de outros *softwares* sobre ele (ex: QEMU). Na virtualização, o código é executado em uma máquina virtual, que é um *software* que simula um ambiente operacional completo que se comporta como se fosse um computador independente (ex: VirtualBox, VMWare e VirtualPC).

Limitações das técnicas comumente utilizadas para interceptar as chamadas de sistema ou instrumentar o ambiente do *sandbox* podem levar à evasão da análise por exemplares de *malwares* modernos. Isso, como afirma (O'KANE, 2011), é agravado pelo fato de alguns exemplares de malware terem por característica a mutação de seus comportamentos durante execuções distintas, fazendo com que o relatório gerado por um sistema possa ser diferente do relatório gerado por outro, no que diz respeito às ações efetuadas pelo *malware* sob análise. Técnicas de evasão de análise são descritas no capítulo 4 desse trabalho.

3.4 ANÁLISE AUTOMÁTICA DE CÓDIGO MALICIOSO

A análise automática de *malware* consiste em observar as características funcionais do *malware* através da sua execução num ambiente controlado.

Segundo (WILLEMS, 2007), as principais metodologias nesse tipo de análise baseiam-se em: (a) comparação do *status* do sistema operacional antes e imediatamente após a execução do artefato; e (b) monitoramento das ações em tempo de execução. Na primeira abordagem, busca-se fazer uma comparação do sistema operacional completo identificando alterações causadas pelo arquivo binário executado. Como resultado, essa técnica traça uma visão geral das funcionalidades do binário, como arquivos criados, dados removidos, entre outros. Essa solução, entretanto, não determina mudanças dinâmicas intermediárias ao estado inicial e final da comparação do sistema. Mudanças como a criação de arquivos durante a execução e a remoção de arquivos antes do final do processo são invisíveis a essa análise. Por outro lado, na segunda abordagem, cuja monitoração das ações do *malware*

é dada durante a execução, tais ações são registradas. A implementação é feita através de *hooks* (ganchos) como DLL *Injections* (SIKORSKI, 2012). Mesmo sendo mais complexa de implementar, a analise de binários durante sua execução, vem popularizando-se devido ao bom resultado da técnica perante códigos polimórficos e ofuscados.

De forma geral, a principal limitação da análise dinâmica de *malware* é a possibilidade de executar apenas uma amostra de binário de cada vez. Afinal, a execução de outros binários no mesmo ambiente controlado dificulta a distinção das ações de cada *malware*.

A facilidade de reconstruir um ambiente virtualizado propiciou o surgimento de ferramentas mais detalhistas e escaláveis para a análise dinâmica como é o caso dos *sandboxes*. Ferramentas como CWSandbox, Anubis, Cuckoo, Norman Sandbox, ThreatExpert, Joebox e CaptureBat são automatizadas para análise dinâmica de arquivos binários. Essas ferramentas caracterizam-se por executar um arquivo num ambiente controlado registrando em forma de relatório as ações realizadas pelos *malwares*. Como característica, os *sandboxes* tradicionalmente simulam o sistema operacional *Windows*, já que a grande maioria de *malwares* existentes é escrita para o mesmo (BARFORD, 2007). Funcionalidades de um *sandbox* incluem o monitoramento de:

- arquivos criados ou modificados;

- acessos ou modificações da chave do registro do sistema;

- bibliotecas dinâmicas carregadas;

- áreas da memória virtual acessadas;

- processos criados;

- conexões de rede instanciadas e

- dados transmitidos pela rede.

Segundo (WILLEMS, 2007), os relatórios disponibilizados pelos *sandboxes* podem variar bastante devido a particularidades de implementação dos mesmos. Por exemplo, o *sandbox* Norman simula um computador conectado na rede reimplementando partes do núcleo do sistema Windows emulado. Já o *sandbox* Anubis é implementado com base no sistema Qemu. Além das características inerentes às particularidades do sistema emulado, os *sandboxes* especificam um conjunto limitado de chamadas de sistemas a ser monitorada.

O maior problema da grande maioria dos *sandboxes* é o fato dos mesmos serem comerciais e de não serem abertos para modificações e estudo. Muitos *sandboxes* são concebidos como projetos de pesquisa dentro de universidades e comunidades de Segurança da

Informação. Quando esses projetos atingem um maior reconhecimento do mercado geralmente são comprados por grandes empresas de TI e se tornam produtos comercias com licenças pagas. Caso mais recente foi do CWSandbox que passou a se chamar GFISandbox depois de comprado pela empresa GFI Software. Nesse trabalho, foi escolhido para estudo e adequação às necessidades da pesquisa o *Cuckoo Sandbox* que está sob licença GNU GPL.

3.4.1 CUCKOO SANDBOX

A FIG. 3.5 mostra o processo de análise do artefato fotos.exe. O arquivo contém o *malware* Trojan.Banker.BAT.Qhost.al.

FIG. 3.5: Tela do *Cuckoo Host* durante o processo de análise do arquivo fotos.exe.

Esse *sandbox* começou como um projeto desenvolvido durante o *Google Summer of Code* 2010 dentro da organização do projeto *Honeynet* (HONEYNET, 2012). As ideias do desenvolvimento de *Cuckoo* são:

- criar um produto *Open Source* para análise de artefatos;

- ser um instrumento capaz de analisar qualquer tipo de arquivo malicioso;

- fornecer um *sandbox*, que pode ser configurado para ser executado tanto em máquinas virtuais quanto em máquinas reais;

- torná-lo capaz de ser distribuído livremente.

A FIG. 3.6 mostra a coleção de arquivos gerados durante o processo de análise do *malware* Net-Worm.Win32.Kolabc.buj.

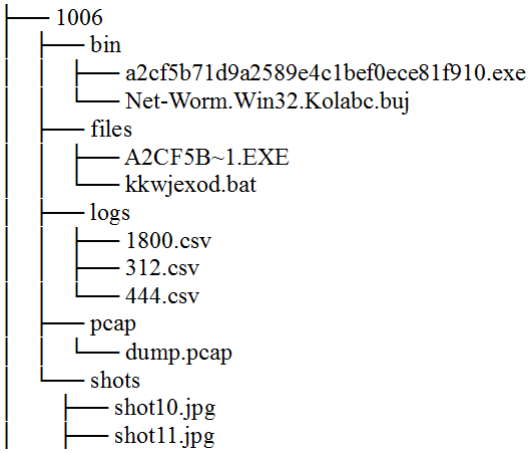

FIG. 3.6: Arquivos gerados pelo *Cuckoo* ao analisar o *malware* Net-Worm.Win32.Kolabc.buj.

O relatório gerado pelo *Cuckoo* contém :

- chamadas API relevantes do SO *Windows* e rastreamento de todos os processos gerados recursivamente;

- *dump* do tráfego de rede gerado durante a execução do *malware*;

- arquivos que foram baixados e excluídos durante a execução;

- imagens (*snapshots*) das telas geradas durante todo o processo de análise.

A FIG. 3.7 mostra uma pequena parte do relatório gerado após o processo de análise do *malware* Net-Worm.Win32.Kolabc.buj.

Abaixo, estão destacadas algumas vantagens da utilização do *Cuckoo Sandbox*:

- os dados capturados são de funções de alto nível, facilitando a leitura dos relatórios;

- código aberto disponível para customização;

- contém mecanismos que dificultam a detecção do sistema;

- suporta análise em qualquer versão do SO *Windows*;

- ao contrário de muitos *sandboxes*, que só analisam arquivos no formato PE (*Portable Executable*), o *Cuckoo* analisa vários tipos de arquivos (ex: DLL, PDF, MSOffice e scripts PHP. A nova versão 0.5 analisa também jar, applet e zip);

- gera relatórios nos formatos CSV, HTML, XML, JSON e MongoDB.

"20121105010857.422","1800","a2cf5b71d9a2589e4c1bef0ece81f910.exe","ShellExe
cuteExW","lpVerb->open","lpFile->kkwjexod.bat","lpParameters-
>(null)","lpDirectory->(null)","hProcess->0x00000000"
"20121105010857.603","1800","a2cf5b71d9a2589e4c1bef0ece81f910.exe","RegOpen
KeyW","hKey->HKEY_LOCAL_MACHINE","lpSubKey-
>Software\Policies\Microsoft\Windows\Safer\CodeIdentifiers","phkResult-
>0x0012d634"
"20121105010857.613","1800","a2cf5b71d9a2589e4c1bef0ece81f910.exe","RegOpen
KeyW","hKey->HKEY_CURRENT_USER","lpSubKey-
>Software\Policies\Microsoft\Windows\Safer\CodeIdentifiers","phkResult-
>0x0012d634"
"20121105010857.783","1800","a2cf5b71d9a2589e4c1bef0ece81f910.exe","CreatePr
ocessA","lpApplicationName->(null)","lpCommandLine-
>C:\WINDOWS\system32\csrs.exe","dwProcessId->312"

FIG. 3.7: Parte do relatório gerado pelo *Cuckoo* ao final da analise do *malware* Net-Worm.Win32.Kolabc.buj.

- analisa mais de um *malware* ao mesmo tempo;

- evita o tempo gasto com a inicialização do *Windows*, pois as máquinas virtuais podem ser configuradas para inicializarem depois desse ponto;

- gera imagens (*snapshots*) durante todo o processo de análise;

Abaixo, estão destacadas algumas desvantagens da utilização do *Cuckoo Sandbox*:

- existem componentes do sistema de análise dentro do ambiente de análise;

- é necessário preparar o ambiente para executar a análise.

3.5 APRENDIZADO DE MÁQUINA

(MITCHELL, 1997) define aprendizado como:

"A capacidade de melhorar o desempenho na realização de alguma tarefa por meio da experiência."

Técnicas de aprendizado de máquina (AM) estão relacionadas com programas computacionais que melhoram seu desempenho automaticamente através da experiência. Os métodos de aprendizado de máquina têm sido utilizados em diversas aplicações como veículos autônomos que aprendem a dirigir em vias expressas, reconhecimento da fala, detecção de fraudes em cartões de crédito, estratégias para a construção de jogos, programas de mineração de dados que descobrem regras gerais em grandes bases de dados,

entre outros.

3.5.1 ALGORITMOS

Algoritmos de AM aprendem automaticamente a partir de um conhecimento ou experiência gerados (MITCHELL, 1997). Um programa treinador utiliza algoritmos de AM para ser capaz de criar respostas úteis para novos casos de teste a partir de uma generalização das respostas de casos conhecidos. O aprendizado de máquina pode ser dividido em supervisionado, não supervisionado, semi-supervisionado ou aprendizado por reforço.

Algoritmo de aprendizado supervisionado gera uma função que mapeia entradas em saídas desejadas (tipicamente usado em problemas de classificação). Diferentemente, num algoritmo de aprendizado não supervisionado, o próprio sistema aprendiz modela o conjunto de entradas (tipicamente usado em problemas de agrupamento). Como um meio termo, aparecem os de aprendizado semi-supervisionado, que combinam tanto o supervisionado, quanto o não supervisionado para gerar uma função apropriada ou um classificador. Por sua vez algoritmos de aprendizado por reforço aprendem como agir dada uma observação do ambiente, isto é, utilizam os retornos dados pelo ambiente, em cada observação, como um guia para a aprendizagem. Um exemplo da aplicação dos algoritmos de aprendizado por reforço é a de ensinar um robô a encontrar a melhor trajetória entre dois pontos, onde os algoritmos punem a passagem por trechos pouco promissores e recompensam a passagem por trechos promissores (FACELI, 2011).

O presente trabalho utiliza apenas técnicas de aprendizado supervisionado.

Nas próximas sub-seções, são apresentadas algumas das principais abordagens usadas na resolução de problemas por meio de aprendizado de máquina.

3.5.2 NAIVE BAYES

O Classificador *Naive Bayes* é provavelmente um dos classificadores mais utilizados em aprendizado de máquina (ZHANG, 2004). O classificador é denominado ingênuo (*naive*) por assumir que os atributos são condicionalmente independentes, ou seja, a informação de um evento não é informativa sobre nenhum outro. Apesar desta premissa "ingênua" e simplista, o classificador reporta um bom desempenho em várias tarefas de classificação. Este fenômeno é discutido em (MCCALLUM, 1998) e (CHAKRABARTI, 2002).

Até as primeiras décadas do século XVIII, problemas relacionados a probabilidade de certos eventos, dadas certas condições, estavam bem resolvidos. Por exemplo, dado um número específico de bolas negras e brancas em uma urna, qual é a probabilidade de eu sortear uma bola preta? Tais problemas são chamados de problemas de probabilidade condicional. Porém, o problema inverso começou a chamar a atenção dos matemáticos da época: dado que uma ou mais bolas foram sorteadas, o que pode ser dito sobre o número de bolas brancas e pretas na urna?

Thomas Bayes, um ministro presbiteriano inglês do século XVIII, foi o primeiro a formalizar uma teoria para problemas desta natureza. Teoria vista como revolucionária no meio científico da época (WIKIPEDIA, 2012). É exatamente este pensamento inverso que se busca ao treinar um classificador com esse algoritmo. Dado que tenho exemplos de cada classe. O que posso inferir sobre o processo gerador destas classes?

3.5.3 SVM

Os algoritmos de aprendizado de máquina têm como objetivo a determinação de limites de decisão que produzam uma separação ótima entre classes por meio da minimização dos erros (VAPNIC, 1995).

SVM (*Support Vector Machine*) consiste em uma técnica computacional de aprendizado para problemas de reconhecimento de padrões. Introduzida por meio da teoria estatística de aprendizagem, essa classificação é baseada no princípio de separação ótima entre classes, tal que se as classes são separáveis, a solução é escolhida de forma a separar ao máximo as classes.

Recentemente, o SVM foi utilizado na área de sensoriamento remoto com relativo sucesso (BROWN, 2000) e (MELGANI, 2004).

3.5.4 ÁRVORES DE DECISÃO

A partir da idéia inicial de Hunt (QUINLAN, 1993), no final da década de 50, as árvores de decisão foram usadas com sucesso em sistemas de aprendizado de máquina e têm sido estudadas tanto na área de reconhecimento de padrões quanto na área de aprendizado de máquina. A partir disso, outros pesquisadores trabalharam em métodos similares, como (BREIMAN, 1984), que desenvolveram os fundamentos do sistema CART, utilizado por Quinlan como base para o sistema ID3 (DAFONSECA, 1994) e, posteriormente, no sistema C4.5 (QUINLAN, 1993).

Árvore de decisão é uma das principais técnicas de aprendizado de máquina, especialmente quando problemas de classificação envolvem atributos nominais.

Uma árvore de decisão é composta por três elementos básicos:

- Nó raiz: corresponde ao nó de decisão inicial que normalmente é gerado utilizando-se o atributo mais discriminante entre as classes envolvidas no problema;

- Arestas: correspondem aos diferentes valores possíveis das características;

- Nó folha: corresponde a um nó de resposta, contendo a classe a qual pertence o objeto a ser classificado.

Em árvores de decisão, duas grandes fases devem ser asseguradas:

1. Construção da árvore: uma árvore de decisão é construída com base no conjunto de dados de treinamento, sendo dependente da complexidade dos dados. Uma vez construída, regras podem ser extraídas através dos diversos caminhos providos pela árvore para que sejam geradas informações sobre o processo de aprendizado.

2. Classificação. Para classificar uma nova instância, os atributos da amostra são testados pelo nó raiz e pelos nós subsequentes, caso seja necessário. O resultado deste teste permite que os valores dos atributos da instância dada sejam propagados do nó raiz até um dos nós folhas, ou seja, até que uma classe seja atribuída à amostra.

Para melhor ilustrar o processo de classificação de uma árvore de decisão, a FIG. 3.8 apresenta um modelo utilizado no treinamento de dados para a tarefa de detecção de malware. Inicialmente, uma base de dados supervisionada é submetida ao algoritmo, que por sua vez, gera uma árvore construída para separar amostras da classe "benigno" de amostras da classe "maligno". Após a construção da árvore, dados desconhecidos podem ser submetidos ao classificador, o qual atribuirá um das duas classes à amostra testada.

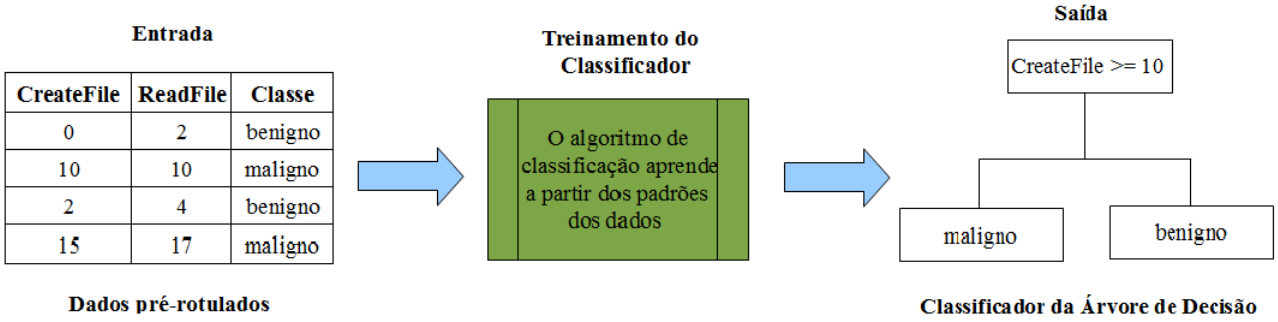

FIG. 3.8: Treinamento do Classificador de Árvore de Decisão.

3.5.4.1 ID3

O ID3 (*Iterative Dichotomiser 3*) (QUINLAN, 1986) é o algoritmo pioneiro em indução de árvores de decisão. Ele é um algoritmo recursivo e baseado em busca gulosa, procurando, sobre um conjunto de atributos, aqueles que melhor dividem os exemplos, gerando sub-árvores.

A principal limitação do ID3, em sua implementação original, é que ele só lida com atributos categóricos discretos, não sendo possível apresentar a ele conjuntos de dados com atributos contínuos, por exemplo. Nesse caso, os atributos contínuos devem ser previamente discretizados. Além dessa limitação, o ID3 também não apresenta nenhuma forma para tratar valores desconhecidos, ou seja, todos os exemplos do conjunto de treinamento devem ter valores conhecidos para todos os seus atributos.

É de conhecimento geral que, na prática, os conjuntos de dados possuem muitos valores desconhecidos. Logo, para se utilizar o ID3, é necessário gastar um bom tempo com pré-processamento dos dados.

O ID3 utiliza o ganho de informação para selecionar a melhor divisão. No entanto, esse critério não considera o número de divisões (número de arestas), e isso pode acarretar em árvores mais complexas. Somado a isso, o ID3 também não apresenta nenhum método de pós-poda, ou seja, não transforma em nós folha aqueles ramos que não apresentam nenhum ganho significativo, o que poderia amenizar esse problema nas árvores mais complexas.

3.5.4.2 C4.5

O algoritmo C4.5 (QUINLAN, 1993) representa uma significativa evolução do ID3 (QUINLAN, 1986). As principais contribuições em relação ao ID3 são:

- lida tanto com atributos categóricos (ordinais e discretos) como com atributos contínuos. Para lidar com atributos contínuos, o algoritmo C4.5 define um limiar e então divide os exemplos de forma binária: aqueles cujo valor do atributo é maior que o limiar e aqueles cujo valor do atributo é menor ou igual ao limiar;

- trata valores desconhecidos. O algoritmo C4.5 permite que os valores desconhecidos para um determinado atributo sejam representados como '?', e o algoritmo trata esses valores de forma especial. Esses valores não são utilizados nos cálculos de ganho e entropia;

- utiliza a medida de razão de ganho para selecionar o atributo que melhor divide os exemplos. Essa medida se mostrou superior ao ganho de informação, gerando árvores

mais precisas e menos complexas;

- lida com problemas em que os atributos possuem custos diferenciados;

- apresenta um método de pós-poda das árvores geradas. O algoritmo C4.5 faz uma busca na árvore, de baixo para cima, e transforma em nós folha aqueles ramos que não apresentam nenhum ganho significativo.

A ferramenta de mineração de dados WEKA (WITTEN, 2011) disponibiliza a implementação do algoritmo C4.5, porém o mesmo é chamado de J48 nessa ferramenta. O C4.5 é um dos algoritmos mais utilizados na literatura, por ter mostrado ótimos resultados em problemas de classificação. Embora já tenha sido lançado o C5.0, o C4.5 possui código-fonte disponível, enquanto que o C5.0 é um software comercial (QUINLAN, 2012).

O C4.5 é do tipo:

- Guloso: executa sempre o melhor passo avaliado localmente, sem se preocupar se este passo, junto à sequência completa de passos, vai produzir a melhor solução ao final;

- Dividir para conquistar: partindo da raiz, criam-se subárvores até chegar nas folhas, o que implica em uma divisão hierárquica em múltiplos sub-problemas de decisão, os quais tendem a serem mais simples que o problema original.

3.5.4.3 CART

O algoritmo CART (*Classification and Regression Trees*) foi proposto em (BREIMAN, 1984) e consiste de uma técnica não-paramétrica que induz tanto árvores de classificação quanto árvores de regressão, dependendo se o atributo é nominal (classificação) ou contínuo (regressão).

Dentre as principais virtudes do CART está a grande capacidade de pesquisa de relações entre os dados, mesmo quando elas não são evidentes, bem como a produção de resultados sob a forma de árvores de decisão de grande simplicidade e legibilidade (DAFONSECA, 1994).

As árvores geradas pelo algoritmo CART são sempre binárias, as quais podem ser percorridas da sua raiz até as folhas respondendo apenas a questões simples do tipo "sim" ou "não".

Os nós que correspondem a atributos contínuos são representados por agrupamento de valores em dois conjuntos. Da mesma forma que no algoritmo C4.5, o CART utiliza a técnica de pesquisa exaustiva para definir os limiares a serem utilizados nos nós para

dividir os atributos contínuos. Adicionalmente, o CART dispõe de um tratamento especial para atributos ordenados e também permite a utilização de combinações lineares entre atributos (agrupamento de valores em vários conjuntos).

Diferente das abordagens adotadas por outros algoritmos, os quais utilizam pré-poda, o CART expande a árvore exaustivamente, realizando pós-poda por meio da redução do fator custo-complexidade (BREIMAN, 1984). Segundo os autores, a técnica de poda utilizada é muito eficiente e produz árvores mais simples, precisas e com boa capacidade de generalização.

3.5.4.4 RANDOM FOREST

A técnica de *Random Forest* (BREIMAN, 2001) pode ser entendida como uma extensão da técnica de árvores de decisão, não no sentido de que seja mais apropriada para conjuntos de dados maiores, mas sim por se tratar de um procedimento que faz uso de métodos de reamostragem a fim de melhorar a precisão dos modelos construídos. De acordo com (SÁ LUCAS, 2011), tal técnica consiste, essencialmente, em:

- *Bagging*: gerar, através de reamostragem, um conjunto de subamostras provenientes da amostra original, selecionadas de maneira aleatória simples com reposição;

- *Boosting*: construir, para cada subamostra, um modelo de árvore de decisão, aumentando a ponderação das observações incorretamente classificadas com base nos modelos criados para as outras subamostras; e

- *Randomizing*: desenvolver os modelos de árvore de decisão utilizando, em cada nó, um conjunto de variáveis selecionadas aleatoriamente, diferente para cada nó.

3.5.5 FRAMEWORKS DE APRENDIZADO DE MÁQUINA

3.5.5.1 WEKA

Weka (*Waikato Environment for Knowledge Analysis*) é um pacote desenvolvido pela Universidade de Waikato, em 1993, com o intuito de agregar algoritmos para mineração de dados na área de Inteligência Artificial. O software é licenciado pela *General Public License* sendo, assim, possível a alteração do seu código-fonte. Weka é elaborado em linguagem Java. A FIG. 3.9 mostra uma instanciação do Framework Weka para experimentação.

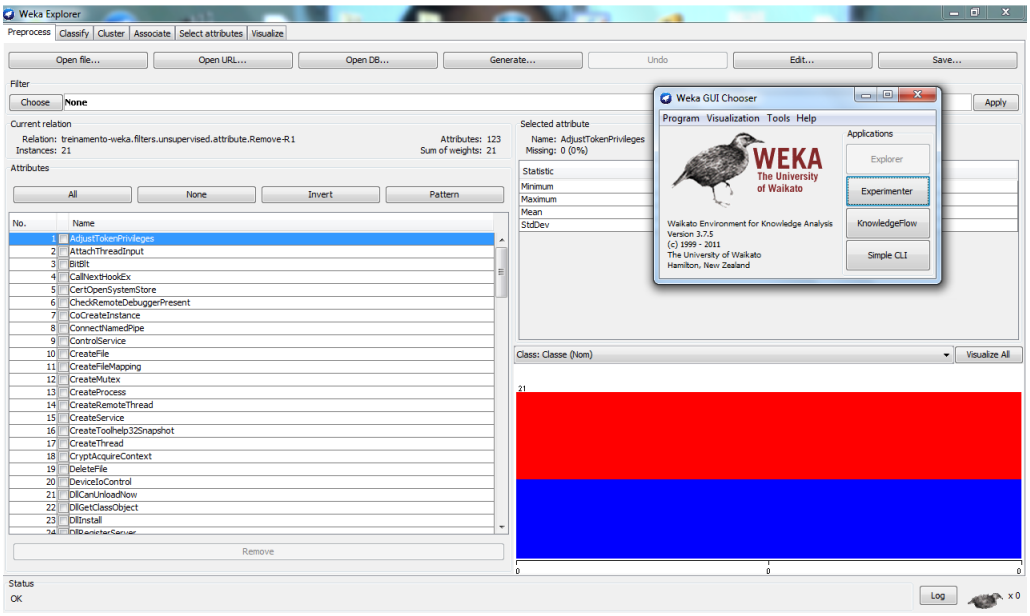

FIG. 3.9: Uma instanciação do Framework Weka para experimentação.

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. Os algoritmos podem ser aplicados diretamente a um conjunto de dados ou chamados a partir do seu próprio código Java (FIG. 3.10).

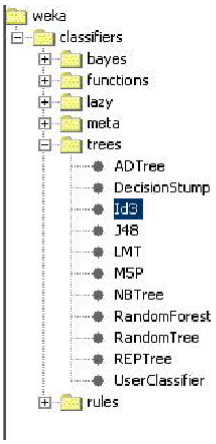

FIG. 3.10: Seleção do Algoritmo no Weka.

Contém 10 bases de dados prontas para serem mineradas e testadas. A visualização das Árvores de Decisões já ocorre com a poda, transformando em nós folha aqueles ramos que não apresentam nenhum ganho significativo, e a Matriz de Confusão, que é uma tabela que permite visualizar o desempenho de um algoritmo, é apresentada apontando

os erros e acertos considerados pelo sistema.

Por ter sido desenvolvido usando a abordagem de *framework*, WEKA é extensível, permitindo, portanto, que novos algoritmos ou funcionalidades sejam adicionadas de maneira relativamente confortável.

Vale ressaltar que para que uma base de dados seja carregada no software, é necessário que o arquivo esteja no formato .arff (Attribute-Relation File Format), formato de leitura para o Weka.

As suas características, bem como as técnicas nele implementadas são descritas de forma detalhada em (WITTEN, 2011), cujos autores são os responsáveis pela implementação da ferramenta. O software está disponível para *Windows*, *Linux* e outras plataformas.

Finalmente, devido a todas essas características acima descritas, esta ferramenta foi a escolhida para os testes iniciais nesse trabalho.

3.5.5.2 FAMA

O FAMA (FRAMEWORK DE APRENDIZADO DE MÁQUINA) foi desenvolvido inicialmente com a função de classificar textos usando algoritmos de AM. Para isso, o framework recebe um arquivo de texto já tokenizado (corpus) e com uma classificação prévia de cada token executada por um sistema especialista de base humana (base do aprendizado).

A produção do FAMA ocorreu na plataforma C++, onde foram criadas classes especializadas em determinadas tarefas e a execução do programa se dá pela troca de mensagens entre as classes. A FIG. 3.11 mostra um diagrama de classes da estrutura básica do framework, apresentando as classes abstratas Avaliador, Treinador, Classificador e Corpus.

A Corpus é a classe que armazena o texto na qual se deseja aplicar os métodos de AM. Ela contém dois métodos abstratos: carregarArquivo e gravarArquivo, que servem para carregar o arquivo texto da memória secundária para a principal e da principal para a secundária, respectivamente. Esses métodos podem ser implementados por classes herdeiras, especializadas em tipos diferentes de arquivos. No FAMA, correntemente, há somente a classe CorpusMatriz que trabalha com arquivos de texto. Uma classe que trabalha com outro tipo de arquivo poderia ser adicionada ao projeto facilmente, bastando acoplá-la à classe Corpus.

A Avaliador é implementada pela classe AvaliadorAcuracia e seu método (calcularDe-

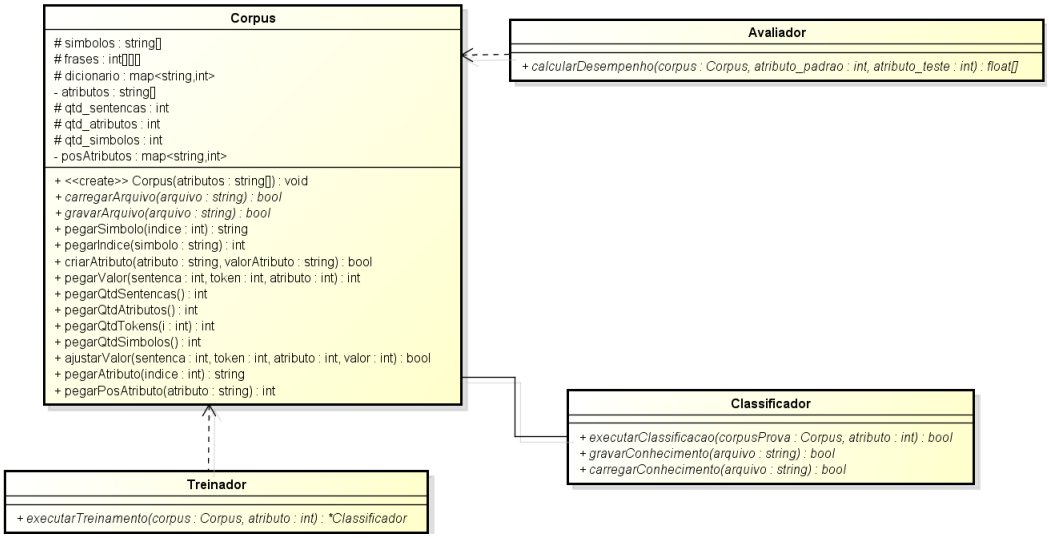

FIG. 3.11: Classes abstratas do FAMA.

sempenho) é o responsável por fazer uma avaliação do resultado final da classificação do corpus. A AvaliadorAcuracia avalia com base na porcentagem de acertos do Classificador. O FAMA possibilita a utilização de outro critério avaliador, bastando uma nova classe implementar a classe Avaliador com o seu método específico.

A FIG. 3.12 apresenta as duas implementações citadas (CorpusMatriz e AvaliadorAcuracia), essa instância está presente em vários algoritmos de classificação.

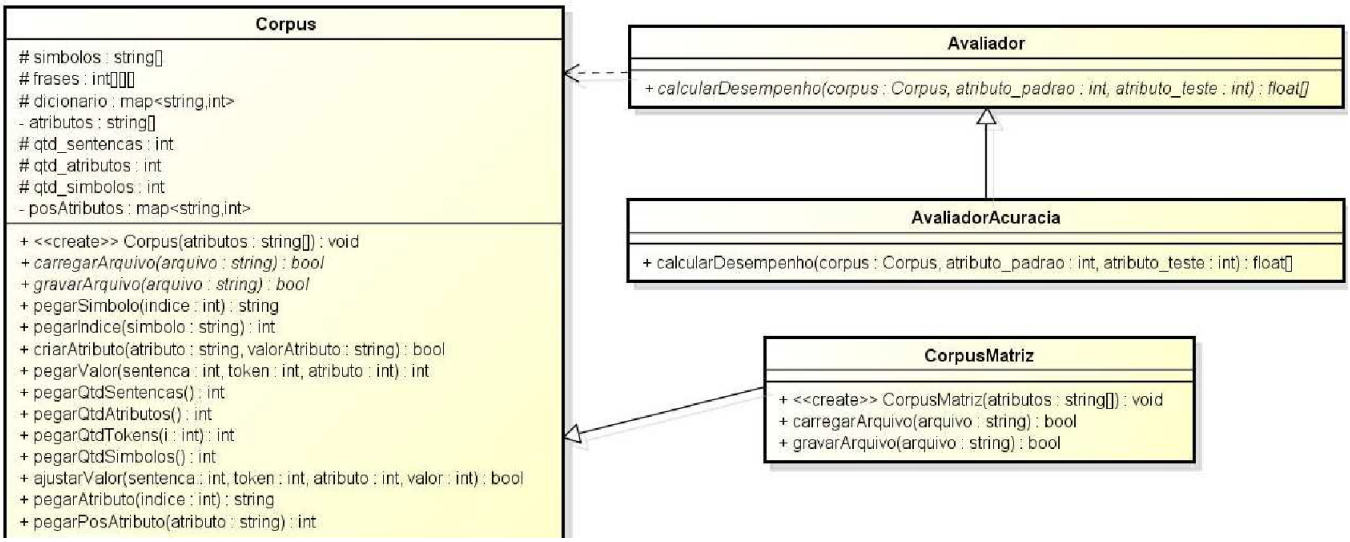

FIG. 3.12: Instância da Corpus e Avaliador utilizadas no FAMA.

Dentro da classe Corpus, os dados são armazenados em uma matriz, em que cada elemento é representado por uma tupla. O primeiro elemento é um token do texto e os

seguintes são os diferentes tipos de classificação para aquele token, ou então, os mesmos tipos de classificações executadas por diferentes classificadores. A fim de diminuir a quantidade de memória utilizada e agilizar processos de comparação de palavras, os elementos de cada tupla são representados por números inteiros que tem uma relação biunívoca com cada palavra do dicionário (diferentes strings levam em números diferentes e vice-versa).

Uma vez carregado o arquivo texto por um objeto da classe herdeira de Corpus, esse é então passado para um objeto da classe Treinador, essa classe é a especialista em executar os algoritmos de AM, ou seja, é nessa etapa que o programa aprende a classificar um texto. A classe Treinador tem o método abstrato executarTreinamento que é implementado pelas classes derivadas dele, cada uma contendo um método diferente de aprendizado. Atualmente, esse três métodos do FAMA são implementados em alguns algoritmos como HMM, TBL, regresão logítica, regresão linear, *Decision Stump* e, ID3.

O resultado da Treinador é então utilizado na Classificador afim de classificar outro conjunto de dados qualquer armazenado em um objeto de uma classe herdeira da Corpus.

4 TÉCNICAS ANTI-ANÁLISE

Para tentar lidar com a quantidade excessiva de novos exemplares que surgem diariamente, organizações que trabalham com antivírus necessitam compreender o funcionamento dos novos exemplares de *malware* a fim de fazer assinaturas para os mesmos sob demanda, visando garantir a qualidade do seu produto em identificar programas maliciosos.

Em contrapartida à ação das empresas de antivírus, os criadores de *malwares* utilizam diversas técnicas para evitar que a análise seja feita de forma completa. Entre essas técnicas estão o empacotamento de executável e *anti-dumping*, que servem para evitar que a assinatura do artefato seja encontrado e a imagem do programa em execução não possa ser copiada da memória. Noutras técnicas alguns artefatos detectam a presença de depuradores ou se estão sendo executados em uma máquina virtual, e então encerram seu funcionamento ou realizam outras operações específicas.

Estas técnicas dificultam a análise do artefato, porém existem métodos que permitem contornar estes problemas como, por exemplo, evitar que a máquina virtual e o depurador sejam detectados e utilizar desempacotadores de códigos executáveis.

4.1 EMPACOTAMENTO

A abordagem mais adotada por pesquisadores e empresas de segurança é a análise minuciosa dos artefatos através de engenharia reversa, para que assim medidas de prevenção por meio de assinaturas de antivírus e atualizações de segurança dos *softwares* sejam efetuadas rapidamente. Porém essa abordagem tem uma grande dificuldade que é o uso de *packers* pelos *malwares*. *Packers* ou empacotadores são módulos que ofuscam o código *assembly* a fim de dificultar a análise dos artefatos (DAVIS, 2009).

Atualmente, o empacotamento de executáveis é utilizado unindo-se o código de descompressão e o próprio programa em um único arquivo executável. Além de diminuir o tamanho dos arquivos e proteger programas da engenharia reversa, o empacotamento é utilizado para ocultar artefatos para que não sejam identificados. Alguns compiladores também utilizam tecnologias que empacotam o executável gerado. As ferramentas maliciosas são comumente empacotadas para dificultar a sua identificação por um antivírus

quando elas se instalam em uma máquina alvo. Outra finalidade do empacotamento é a ocultação do modo de funcionamento do artefato, caso a ferramenta maliciosa seja capturada e submetida a uma análise detalhada. Para desempacotar o artefato é utilizado algum unpacker (FIG. 4.1), porém o código fonte pode ter sido empacotado com uma tecnologia avançada, fazendo com que os atuais unpackers não consigam desempacotá-lo. Então deve-se executar o artefato e copiar sua imagem da memória, esta técnica é conhecida como *dumping*.

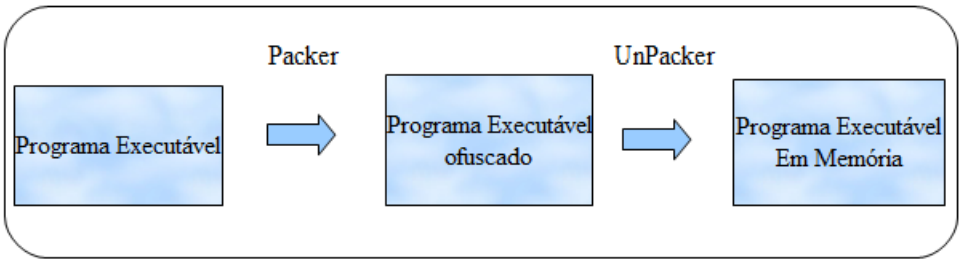

FIG. 4.1: Criação e execução de programas executáveis empacotados.

Existem algumas técnicas para se detectar a presença de packers em códigos executáveis, as mais conhecidas são as que analisam a estrutura de arquivos PE.

4.2 ANTI-DUMPING

Outra tecnologia que vem sendo utilizada são os empacotadores com suporte ao chamado *anti-dumping*. O *dumping* é utilizado para obter o código de um artefato empacotado copiando a sua imagem da memória no momento em que o programa é desempacotado. O objetivo do *anti-dumping* é dificultar a cópia do artefato desempacotado que se encontra na memória. Empacotadores podem tornar o processo de *dumping* menos eficiente das seguintes formas:

- carregando apenas trechos de códigos na memória ao invés de carregar o código completo, evitando assim a obtenção do código *assembly* desempacotado; ou

- excluindo uma seção do código assim que tiver terminado a execução. Esta técnica é conhecida como "*bytes* roubados". Estes *bytes* devem ser restaurados se o programa despejado for executado novamente (BRAND, 2010).

O *anti-dumping* dificulta o uso dos descompiladores e *disassemblers*, pois muitas vezes o código encontrado é muito diferente do código fonte original e muito difícil de entender. Então, para facilitar a análise deste código, são utilizados os depuradores.

4.3 ANTI-DEBUGGING

Com os depuradores (*debuggers*) é possível executar o artefato e verificar todas as instruções que ele está realizando e obter informações como: valores de variáveis em um determinado instante, laços de instruções que o programa entrou e quantas vezes ele os executou, o resultado dos testes condicionais e outras informações. Assim é possível analisar o código detalhadamente e entender sua funcionalidade. Com o objetivo de evitar que o código do artefato seja compreendido são utilizadas técnicas para detectar se o código está em estado de depuração e, caso esteja, o artefato pode encerrar sua execução ou realizar operações diferentes. Segundo (SIKORSKI, 2012), algumas destas técnicas consistem em verificar o valor das seguinte funções da API do *Windows*:

IsDebuggerPresent. Esta é a função API mais simples que existe para detecção de um depurador. Esta função irá retornar zero se você não estiver executando num contexto de um depurador ou um valor diferente de zero se um depurador estiver anexado. O trecho de código mostrado na FIG. 4.2 exemplifica sua utilização.

```
#include <windows.h>
int main() {
 if (IsDebuggerPresent()) printf("Esta sendo depurado");
 else printf("Nao esta sendo depurado");
return 0; }
```

FIG. 4.2: Uso da técnica *anti-debugging* IsDebuggerPresent.

CheckRemoteDebuggerPresent. Esta função API é quase idêntica à IsDebuggerPresent. Apesar do nome, esta função não verifica um depurador em uma máquina remota, mas sim um processo na máquina local. Ela também verifica a estrutura PEB para o campo IsDebugged, no entanto, pode fazê-lo por si mesmo ou por outro processo na máquina local. Esta função recebe um identificador de processo como um parâmetro e irá verificar se esse processo tem um depurador. CheckRemoteDebuggerPresent pode ser usado para verificar o seu próprio processo, simplesmente passando um identificador para o seu processo.

NtQueryInformationProcess. Esta é uma função API nativa da Ntdll.dll que recupera informações sobre um determinado processo. O primeiro parâmetro para esta função é um identificador de processo, o segundo é usado para dizer a função o tipo de informação de processo a ser recuperado. Por exemplo, usando o valor ProcessDebugPort (valor 0x7)

para este parâmetro vai dizer se o processo em questão está sendo depurado. Se o processo não estiver sendo depurado, um zero será devolvido, caso contrário, um número de porta será devolvido.

OutputDebugString. Esta função é utilizada para enviar uma *string* para exibição num depurador. Isto pode ser usado para detectar a presença de um depurador. Por exemplo, o fragmento do programa mostrado na a FIG. 4.3 usa SetLastError para definir o atual código de erro para um valor arbitrário. Se OutputDebugString é chamado e não há depurador, GetLastError não deve mais conter o nosso valor arbitrário, porque um código de erro será definido pela função OutputDebugString se ele falhar. Se OutputDebugString é chamado e há um depurador, a chamada para OutputDebugString deverá ter sucesso, e o valor no GetLastError não deve ser alterado.

```
DWORD errorValue = 12345;
SetLastError(errorValue);

OutputDebugString("Test for Debugger");

if(GetLastError() == errorValue)
{
  ExitProcess();
}
else
{
  RunMaliciousPayload();
}
```

FIG. 4.3: Uso da técnica *anti-debugging* OutputDebugString.

4.4 DETECÇÃO DE EMULADORES E MÁQUINAS VIRTUAIS

A principal desvantagem da utilização de emuladores e máquinas vituais na análise de *malware* é que o código malicioso pode perceber que está em um ambiente emulado ou virtual e, por conta disso, interromper a execução ou apresentar um outro comportamento.

No caso dos emuladores, a detecção pode ser feita de forma muito simples, por exemplo, através da execução de uma instrução no processador que causa um comportamento específico. Uma das formas utilizadas para realizar tal verificação é por meio de *bugs* conhecidos dos processadores de determinadas arquiteturas que fazem com que certas ins-

truções não se comportem como previsto. Se esta instrução for executada no emulador e este não estiver preparado para apresentar o mesmo comportamento de um processador real, o *malware* irá perceber essa diferença, podendo parar ou modificar a sua execução (RAFFETSEDER, 2007).

A detecção de ambiente virtualizado também é simples, com apenas uma instrução *assembly* que, mesmo executada em um nível de baixo privilégio, retorna informações internas sobre o sistema operacional presente na máquina virtual. Tais informações identificam o ambiente virtualizado com base nas diferenças entre estes e sistemas reais (QUIST, 2006). Para combater as técnicas de anti-análise, existem meios de detectar que um *malware* verifica se está em ambiente emulado, ou mesmo de modificar alguns valores presentes no ambiente virtual para tentar disfarçá-lo (KANG, 2009) e (LISTON, 2006). Entretanto, o uso destas técnicas muitas vezes não é suficiente e o *malware* ainda pode detectar que está sendo executado num ambiente emulado/virtual.

4.5 SLEEPING

Como já explicado na subseção 3.3, quando um artefato é submetido a uma análise num *sandbox*, esse artefato será executado por um certo período de tempo, geralmente entre 4 a 5 minutos, e suas ações serão monitoradas de forma automática. Segundo (BRAND, 2010), basta o *malware* esperar para ser executado um período de tempo maior do que o período de tempo da análise para que a técnica de *sandbox* se torne ineficiente. Essa técnica de evasão é chamada de *sleeping*.

Segundo (SHINOTSUKA, 2012), recentemente, os autores de *malware* têm tentado usar outras abordagens desta técnica para enganar os *sandboxes*. Duas dessas abordagens são explicadas abaixo.

Uma técnica de *sleeping* bem interessante é a da execução intervalada, onde as sub-rotinas maliciosas são executados em intervalos específicos. Como mostrado na FIG. 4.4, quando o código é executado, ele aguarda 300.000 milissegundos, ou cinco minutos, antes de executar a subrotina DecryptCode. Espera então 20 minutos e executa a sub-rotina ModifyRegistry. Após a execução da subrotina Network_main, espera mais 20 minutos. Ao fazer pausas entre a execução de cada sub-rotina, a chance do sistema de análise varrer o arquivo precisamente no momento em que ele está executando um pedaço de código malicioso é reduzida.

Outra técnica de *sleeping* mais recente esconde o comportamento malicioso do *malware*

```
thread_network_tasks proc near          ; DATA XREF: function_Launch+7F↓o
                push    ebx
                mov     ebx, eax
                push    300000          ; dwMilliseconds
                call    Sleep
                mov     eax, ebx
                call    DecryptCode___m

loop:                                   ; CODE XREF: thread_network_tasks+28↓j
                push    1200000         ; dwMilliseconds
                call    Sleep
                call    ModifyRegistry___m
                call    network_main
                jmp     short loop
thread_network_tasks endp
```

FIG. 4.4: Malware usando *sleeping* para se evadir da análise.

no movimento do *mouse*. A FIG. 4.5 mostra como isso ocorre. A função API SetWindowsHookEx instala a rotina _main_routine para monitorar o tráfego de mensagens do *mouse* para que quando o *malware* receber mensagens do *mouse*, isto é, se ele for deslocado ou botões clicados, o *malware* é executado. Como uma pessoa normalmente usa um *mouse* quando trabalha com o SO *Windows*, a subrotina _main_routine funciona bem. Mas, como um *sandbox* não usa um *mouse*, o código permanece adormecido e assim não pode ser detectado.

```
push    0               ; dwThreadId
push    0               ; lpModuleName
call    ds:GetModuleHandleA
push    eax             ; hmod
push    offset _main_function ; lpfn
push    WH_MOUSE_LL     ; idHook
call    ds:SetWindowsHookExA
```

FIG. 4.5: *Malware* usando o movimento do mouse para se esconder.

4.6 ANTI-DISASSEMBLY

A base das ferramentas de análise estática de programas é o *disassembly* cuja função é a engenharia reversa de um código binário para recuperar o conjunto de instruções *assembly*, cujo formato é mais compreensível por um humano do que uma sequência de bits. Contudo, como o processo de *disassembly* não é uma ciência exata, ferramentas de análise estática de programas que dependem de sua saída podem gerar resultados incorretos. Estas ferramentas são responsáveis pela abstração dos procedimentos, geração do grafo de fluxo de controle, análise do fluxo de dados e decompilação. Os *disassemblers*

podem ser genericamente divididos em dois tipos: varredura linear e transversal recursivo (LINN, 2003).

Os *disassemblers* de varredura linear começam pelo processamento do segmento de texto (.text) da imagem do código binário obtido no cabeçalho do arquivo e decodifica uma instrução por vez até atingir o final do segmento de texto. Contudo, diante de dados inseridos no segmento de texto o processo de *disassembly* de varredura linear pode ser comprometido, pois qualquer dado neste segmento será interpretado como código.

O código da FIG. 4.6 ilustra a situação em que o dado 'db 0E8h' seria interpretado como se fosse código. Caso os *bytes* coincidissem com alguma codificação de alguma instrução *assembly*, o processo de *disassembly* converteria os *bytes* erroneamente. Em situações de não coincidência, estes *disassemblers* notificam através de pontos de interrogação os *bytes* não convertidos. Para arquiteturas com tamanho variável de instrução, isto ainda significa que o código que segue os dados poderá não ser decodificado corretamente. Uma varredura linear não descobrirá um erro de decodificação até que uma instrução ilegal seja encontrada. Uma propriedade interessante destes *disassemblers* é que eventualmente ocorre a auto sincronização após um determinado número de instruções mal convertidas (LINN, 2003).

```
Main:
        ...
        jmp     Func
        db      0E8h
Func:
        ...
```

FIG. 4.6: Trecho de código contendo dado no segmento de texto de um binário.

Um *disassembler* transversal recursivo visa superar as fraquezas dos *disassemblers* de varredura linear fazendo com que o processo de *disassembly* acompanhe do fluxo de controle do programa. Caso houvesse dados no meio do fluxo de instruções (como o exemplo acima), o *disassembler* acompanharia o salto, e consequentemente, não interpretaria erroneamente o dado 'db 0E8h'. Contudo, o processo de *disassembly* é mais complexo, pois tem que deduzir os possíveis destinos estaticamente, que nem sempre é uma tarefa trivial. Saltos indiretos e *aliasing* são desafios para que esta dedução seja feita corretamente. Uma técnica comum utilizada por códigos maliciosos é explorar estas dificuldades conhecidas.

Por exemplo: existem ferramentas que deliberadamente inserem dados no executável para induzir o *disassembler* a erros (ex: o algoritmo obfLDK) (COLLBERG, 2009).

5 ANÁLISE AUTOMÁTICA DE MALWARE UTILIZANDO SANDBOXES E APRENDIZADO DE MÁQUINA

5.1 DESDOBRAMENTO DA PROPOSTA DE SISTEMA AUTOMÁTICO DE ANÁLISE DINÂMICA DE MALWARE

A FIG. 5.1 detalha o fluxo de análise de *malwares* proposto na seção 1.2.

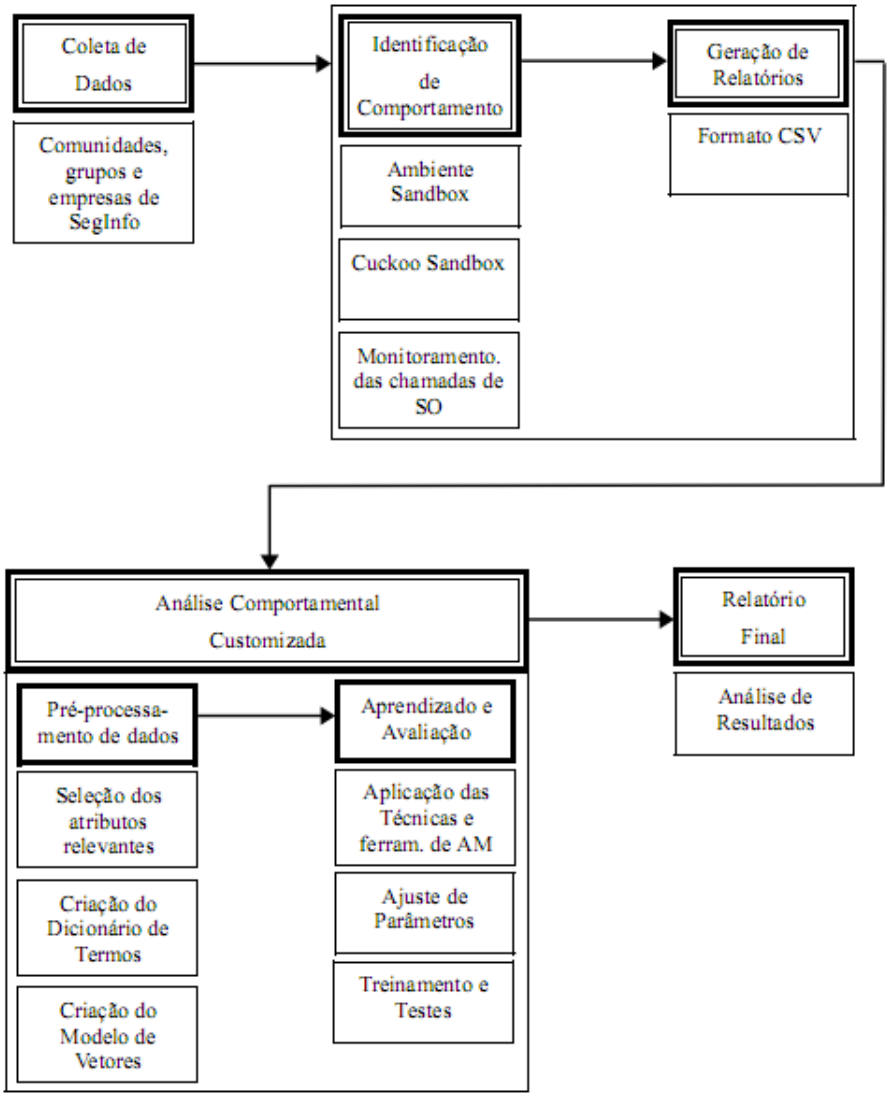

FIG. 5.1: Fluxo de análise de *malware* (detalhado).

5.1.1 COLETA DE DADOS

Nesta fase, são considerados dois conjuntos de exemplos: *malwares* e não *malwares*. Os dois conjuntos estão no formato PE (*Portable Executable*), que é o formato dos arquivos executáveis dos sistemas baseado no Win32 (PIETREK, 1994).

Os malwares foram fornecidos pelo Centro de Tecnologia da Informação Renato Archer (CTI) e pela empresa *BluePex Security Solutions*. E, posteriormente, mais *malwares* foram adquiridos no repositório do site (VXHEAVENS, 2012). Os não *malwares* ou programas benignos foram coletados de máquinas com SO *Windows* limpos.

5.1.2 IDENTIFICAÇÃO DE COMPORTAMENTO E GERAÇÃO DE RELATÓRIOS

É processada a análise automática dos artefatos. Esse processo é feito submetendo-se os exemplos, um de cada vez, de forma automatizada, ao *Cuckoo Sandbox*, gerando um relatório de atividades do artefato, no formato .csv (*Comma-Separated Values*), conforme FIG. 5.2

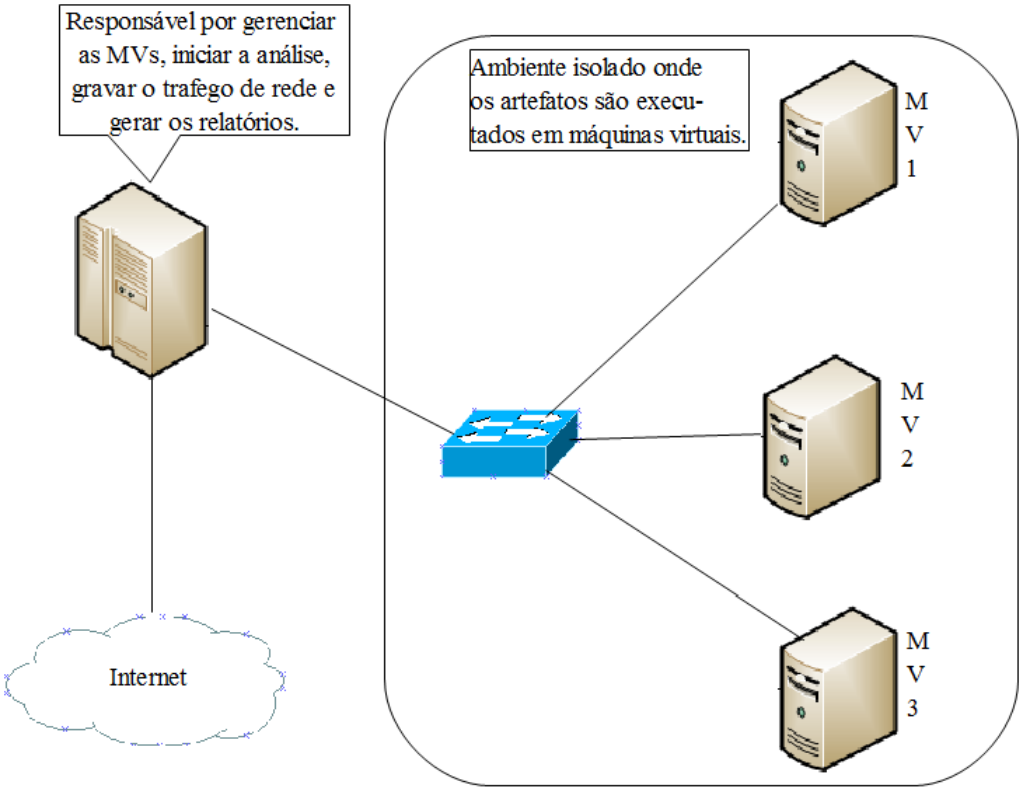

FIG. 5.2: Arquitetura do ambiente de análise do *Cuckoo Sandbox*.

A submissão dos artefatos foi realizada de forma automática através de um *script*

implementado na linguagem *Shell Script*, FIG. 5.3.

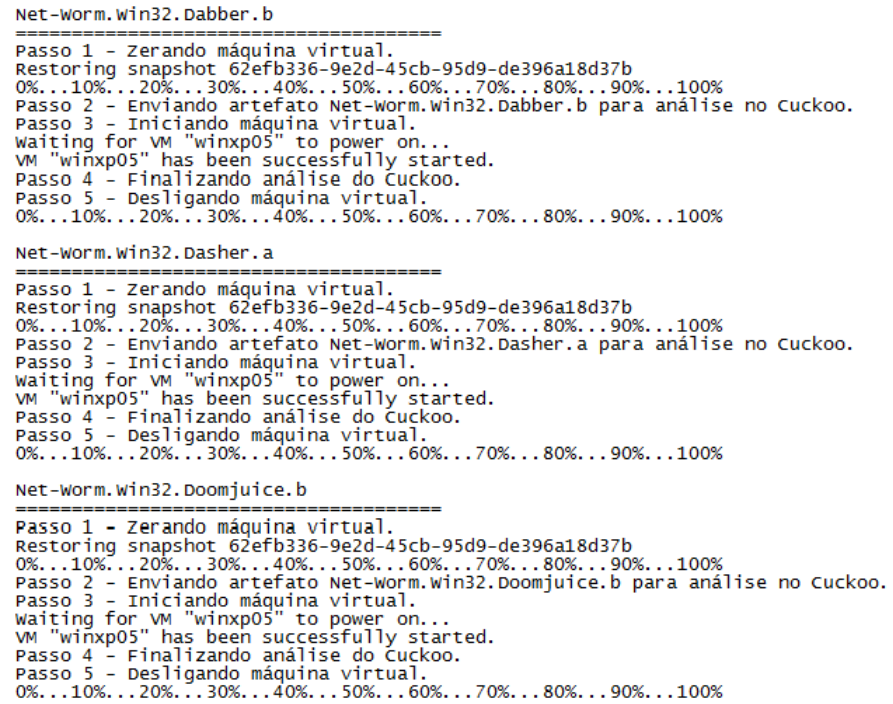

FIG. 5.3: Saida do *Script* de submissão automática de *malwares* para análise do *Cuckoo Sandbox*.

O *script* segue os seguintes passos:

passo 1 - restaura o último *snapshot* salvo, para garantir que a máquina virtual (no caso winxp05) na qual o artefato será analisado esteja limpa;

passo 2 - pega o artefato, no caso o Net-Worm.Win32.Dabber.b, no diretório /home/cuckoo/artefatos/malignos/nao-analisados e envia para a *Cuckoo*. Neste diretório estão todos os *malwares* ainda não analisados;

passo 3 - inicializa a máquina virtual limpa (winxp05) dando início também a analise, pois o *Cuckoo* repassa o artefato para a máquina virtual;

passo 4 - finaliza a análise do *Cuckoo*;

passo 5 - desliga a máquina virtual e se tudo correr bem, move o arquivo analisado do diretório /home/cuckoo/artefatos/malignos/nao-analisados para /home/cuckoo/artefatos/malignos/analisados; e

passo 6 - retorna ao passo 1 e pega o próximo arquivo até o diretório dos arquivos não-analisados ficar vazio.

O processo para cada artefato, da submissão até a geração do relatório do *Cuckoo*, demora aproximadamente 6 minutos independente do hardware utilizado, pois esse tempo é um parâmetro configurável no software (*Cuckoo*).

5.1.3 ANÁLISE COMPORTAMENTAL CUSTOMIZADA

Essa fase, que utiliza técnicas de mineração de dados, é subdividida em: Pré-processamento de Dados e Aprendizado e Avaliação.

5.1.3.1 PRÉ-PROCESSAMENTO DE DADOS

O pré-processamento de dados é feito da seguinte maneira:

1. Seleção dos atributos relevantes

Todos os relatórios .csv passaram por uma seleção, para identificar os atributos mais relevantes.

No experimento, foram selecionados inicialmente 121 atributos, que são algumas APIs comumente utilizadas por *malwares* (SIKORSKI, 2012): AdjustTokenPrivileges, AttachThreadInput, BitBlt, CallNextHookEx, CertOpenSystemStore, CheckRemoteDebuggerPresent, CoCreateInstance, ConnectNamedPipe, ControlService, CreateFile, CreateFileMapping, CreateMutex, CreateProcess, CreateRemoteThread, CreateService, CreateToolhelp32Snapshot, CreateThread, CryptAcquireContext, DeleteFile, DeviceIoControl, DllCanUnloadNow, DllGetClassObject, DllInstall, DllRegisterServer, DllUnregisterServer, EnableExecuteProtectionSupport, EnumProcesses, EnumProcessModules, FindFirstFile, FindNextFile, FindResource, FindWindow, FtpPutFile, GetAdaptersInfo, GetAsyncKeyState, GetDC, GetForegroundWindow, GetKeyState, GetModuleFilename, GetModuleHandle, GetProcAddress, GetStartupInfo, GetSystemDefaultLangId, GetTempPath, GetThreadContext, GetTickCount, GetVersionEx, GetWindowsDirectory, InternetOpen, InternetOpenUrl, InternetReadFile, InternetWriteFile, IsDebuggerPresent, IsNTAdmin, IsWoW64Process, LdrLoadDll, LoadLibrary, LoadResource, LockResource, LsaEnumerateLogonSessions, MapViewOfFile, MapVirtualKey, MmGetSystemRoutineAddress, Module32First, Module32Next, NetScheduleJobAdd, NetShareEnum, NtQueryDirectoryFile, NtQueryInformationProcess, NtQueryInformationThread, NtSetInformationProcess, OleInitialize, OpenFile, OpenMutex, OpenProcess, OpenSCManager, OutputDebugString, PeekNamedPipe, Process32First, Process32Next, QueryPerformanceCounter, QueueUserAPC, ReadFile, ReadProcessMem-

ory, RegCreateKeyEx, RegDeleteKey, RegEnumKeyEx, RegEnumValue, RegisterClassExW, RegisterHotKey, RegisterServiceCtrlHandler, RegOpenKey, ResumeThread, RtlCreateRegistryKey, RtlWriteRegistryValue, SamIConnect, SamIGetPrivateData, SamQueryInformationUse, SetFileTime, SetThreadContext, SetWindowsHookEx, SetWindowTextW, SfcTerminateWatcherThread, WinExec, ShellExecute, ShowWindow, StartServiceCtrlDispatcher, SuspendThread, TerminateProcess, Thread32First, Thread32Next, Toolhelp32Read ProcessMemory, URLDownloadToFile, VirtualAllocEx, VirtualProtectEx, WideCharToMultiByte, WlxLoggedOnSAS, Wow64Disable Wow64FsRedirection, WriteFile, WriteProcessMemory, WSAStartup.

2. Criação do dicionário de termos

Um dicionário de termos contendo os atributos selecionados é criado.

Dessas funções APIs, selecionamos as 20 mais relevantes para compor o nosso dicionário de termos: CreateFile, CreateMutex, CreateProcess, CreateRemoteThread, CreateService, DeleteFile, FindWindow, OpenMutex, OpenSCManager, ReadFile, ReadProcessMemory, RegDeleteKey, RegEnumKeyEx, RegEnumValue, RegOpenKey, ShellExecute, TerminateProcess, URLDownloadToFile, WriteFile, WriteProcessMemory.

Nesse conjunto, foi acrecentado mais dois atributos:

- nr_proc: que se refere a quantidade de processos criados durante a análise.

- nr_down: que se refere a quantidade de downloads realizados durante a análise.

3. Criação do modelo de vetores

Cada relatório .csv é comparado com o dicionário de termos e a frequência de cada termo é registrada, transformando cada relatório .csv em um vetor de atributos. Isto é realizado de forma automática através de um *script*, implementado na linguagem *Shell Script*, que faz o cruzamento do artefato com as funcionalidades mais suspeitas.

A FIG. 5.4 apresenta um exemplo do arquivo de vetores de atributos gerado pelo *script*.

O arquivo com o modelo de vetores é transformado em um arquivo *Attribute-Relation File Format* (ARFF), que é o formato padrão do Weka (HALL, 2009).

A FIG. 5.5 apresenta um exemplo do arquivo de vetores de atributos no formato .arff.

NomeArtefato	CreateFil	CreateMute	CreateRem	ReadFil	RegEnu	RegOpenK	WriteFil	Classe
IM-Worm.Win32.Agent.bw	18	9	2	3	12	7	0	maligno
IM-Worm.Win32.Braban.o	15	6	4	114	0	111	0	maligno
IM-Worm.Win32.Bropia.l	13	6	4	114	0	119	0	maligno
IM-Worm.Win32.Delf.ae	20	9	5	2	12	7	12	maligno
IM-Worm.Win32.Delf.aw	31	9	4	116	12	133	0	maligno
IM-Worm.Win32.Kelvir.ba	13	6	4	114	0	119	0	maligno
IM-Worm.Win32.Licat.ae	6	10	0	0	12	38	0	maligno
IM-Worm.Win32.Sohanad.df	9	10	0	9	0	0	0	maligno
IM-Worm.Win32.VB.au	329	23	15	444	36	382	28	maligno
IM-Worm.Win32.VB.hp	19	6	4	131	0	145	1	maligno
IRC-Worm.DOS.Poison.A	7	0	0	17	0	0	635	maligno
Net-Worm.Win32.Afire.b	21	7	0	90	0	0	824	maligno
Net-Worm.Win32.Agent.d	60	12	1	2	12	21	1	maligno
P2P-Worm.Win32.Agent.bx	13317	323	0	799	0	417	490	maligno
addiag.exe	33	12	0	16	15	29	55	benigno
arquivo.exe	118	26	17	59	32	162	68	benigno
asr_ldm.exe	0	7	0	0	0	0	0	benigno
auditusr.exe	2	0	0	0	0	0	0	benigno
blastcln.exe	109	0	0	0	0	0	206	benigno
cdplayer.exe	59	109	14	439	18	25	98	benigno
certreq.exe	42	12	3	30	0	75	2	benigno
cisvc.exe	1	0	0	0	0	0	0	benigno
ckcnv-2.exe	10	0	0	0	0	0	1	benigno
ckcnv.exe	10	0	0	0	0	0	1	benigno
cleanmgr.exe	39	16	1	16	2	175	2	benigno
clickmein_setup.exe	163	19	325	1106	19	285	173	benigno
cliconfg.exe	10	7	0	0	0	1	0	benigno
clipbrd-2.exe	4	8	0	0	0	0	0	benigno

FIG. 5.4: Exemplo de arquivo de vetores de atributos.

5.1.3.2 APRENDIZADO E AVALIAÇÃO

Nessa fase, técnicas de Aprendizado de Máquina são aplicadas nos arquivos de vetores de atributos para o aprendizado e avaliação de *malwares*. Como os dados foram representados em forma de vetores, diversos algoritmos de classificação podem ser escolhidos e comparados uns com os outros. Na Avaliação é verificado o desempenho da metodologia, levando-se em conta parâmetros como acurácia, falsos positivos e falsos negativos.

Para os testes e experimentos foi utilizado o software Weka 3.7.

A FIG. 5.6 apresenta uma tela do Weka depois de executar o *Random Forest*.

A FIG. 5.7 apresenta a matriz de confusão para um problema de duas classes, onde:

- VP corresponde ao número de verdadeiros positivos, ou seja, o número de exemplos da classe positiva classificados corretamente;

- VN corresponde ao número de verdadeiros negativos, ou seja, o número de exemplos da classe negativa classificados corretamente;

- FP corresponde ao número de falsos positivos, ou seja, o número de exemplos cuja classe verdadeira é negativa mas que foram classificados incorretamente como pertencente

```
@relation malignosebenignos

@attribute CreateFile numeric
@attribute CreateMutex numeric
@attribute CreateProcess numeric
@attribute CreateRemoteThread numeric
@attribute CreateService numeric
@attribute DeleteFile numeric
@attribute FindWindow numeric
@attribute OpenMutex numeric
@attribute OpenSCManager numeric
@attribute ReadFile numeric
@attribute ReadProcessMemory numeric
@attribute RegDeleteKey numeric
@attribute RegEnumKeyEx numeric
@attribute RegEnumValue numeric
@attribute RegOpenKey numeric
@attribute ShellExecute numeric
@attribute TerminateProcess numeric
@attribute URLDownloadToFile numeric
@attribute WriteFile numeric
@attribute WriteProcessMemory numeric
@attribute nr_proc numeric
@attribute nr_down numeric
@attribute Classe {maligno,benigno}

@data
0,3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,maligno
4,3,0,2,0,0,0,3,1,2,0,0,4,7,3,0,0,1,0,0,1,0,maligno
0,3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,maligno
4,3,0,3,0,0,0,0,0,7,0,0,0,0,7,0,0,0,0,0,1,0,maligno
4,3,0,1,1,2,0,0,0,4,6,0,1,7,5,0,0,0,4,0,1,0,maligno
4,3,0,2,0,0,0,0,0,7,0,0,0,0,7,0,0,0,0,0,1,0,maligno
1,3,0,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1,0,maligno
2,0,0,1,0,1,0,0,0,2,0,0,0,0,0,0,0,0,0,1,1,maligno
0,3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,maligno
5,2,0,1,0,1,0,3,1,2,0,0,4,7,3,0,0,0,2,0,1,1,maligno
4,4,1,2,0,0,0,1,0,1,0,0,4,2,5,1,0,0,1,0,1,0,maligno
```

FIG. 5.5: Exemplo de arquivo de vetores de atributos no formato .arff.

à classe positiva;

- FN corresponde ao número de falsos negativos, ou seja, o número de exemplos pertencentes originalmente à classe positiva que foram incorretamente preditos como da classe negativa; e

- $n = VP + VN + FP + FN$.

As Tabelas do capítulo 6 mostram os resultados obtidos através do método proposto. As colunas das tabelas são respectivamente: Classificador, TVP, TFP, TP e ACC, onde segundo (MONARD, 2003):

- TVP (Taxa de Verdadeiros Positivos) corresponde à taxa de acerto na classe positiva:

$$TVP = \frac{VP}{VP + FN}$$

- TFP (Taxa de Falsos Positivos) corresponde à proporsão de exemplos da classe negativa incorretamente classificados pelo classificador/preditor:

$$TFP = \frac{FP}{FP + VN}$$

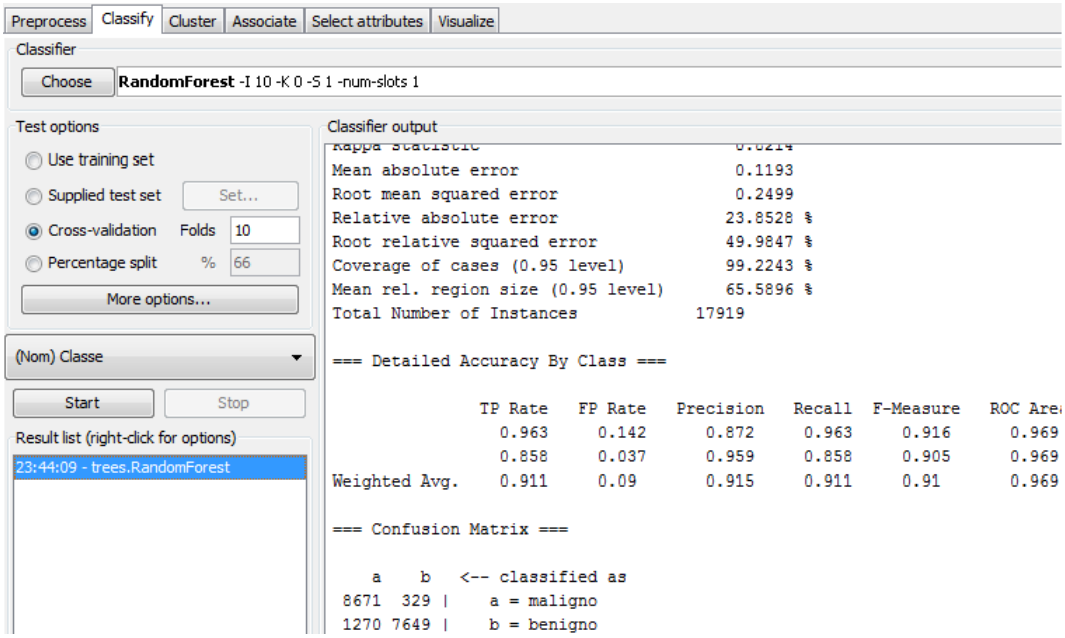

FIG. 5.6: Uma tela do Weka depois de executar o Random Forest.

		Classe predita	
		+	-
Classe verdadeira	+	VP	FN
	-	FP	VN

FIG. 5.7: Matriz de confusão para um problema com duas classes.

- TFN (Taxa de Falso Negativo) corresponde à proporsão de exemplos da classe positiva incorretamente classificados pelo classificador/preditor. Essa taxa não foi calculada pelo Weka:

$$TFN = \frac{FN}{VP + FN}$$

- TP (Taxa de Precisão) corresponde à proporsão de exemplos positivos classificados corretamente entre todos aqueles preditos como positivos pelo classificador/preditor:

$$TP = \frac{VP}{VP + FP}$$

- ACC (Acurácia) corresponde à soma dos valores da diagonal principal da matriz, dividida pela soma dos valores de todos os elementos da matriz:

$$ACC = \frac{VP + VN}{n}$$

6 RESULTADOS EXPERIMENTAIS

Foram realizados 4 experimentos onde os experimentos 01, 02 e 03 analisaram classes diferentes de *malwares* e o experimento 04 analisou um conjunto de malwares composto por uma porção de cada experimento anterior, conforme TAB. 6.1.

TAB. 6.1: Distribuição dos artefatos em cada experimento.

Exp	Nomenclatura	Qtd	Total
01	*worms*	4.617	10.000
	benignos	5.383	
02	*trojans*	11.115	20.034
	benignos	8.919	
03	*backdoors*	9.591	18.510
	benignos	8.919	
04	*worms*	3.000	17.919
	trojans	3.000	
	backdoors	3.000	
	benignos	8.919	

Como explicado na sub-seção 5.1.3.2, os dados são representados em forma de vetores, assim permitindo que diversos algoritmos de classificação possam ser escolhidos e comparados uns com os outros. Em cada experimento, foram executados e comparados os algoritmos mas utilizados pela literatura relacionada, que conforme TAB. 2.1 são:

- *Naive Bayes*;

- SVM;

- J48;

- CART; e

- *Random Forest*.

Para esses experimentos foi utilizado o software Weka 3.7.

Vale ressaltar que para que uma base de dados seja carregada no software, é necessário que o arquivo esteja no formato .arff (*Attribute-Relation File Format*), formato de leitura para o Weka.

6.1 EXPERIMENTO 01

Nesse experimento, 10.000 arquivos (4.617 *malwares* e 5.383 não *malwares*) foram submetidos à análise, conforme tabela TAB. 6.2.

TAB. 6.2: Distribuição dos artefatos do experimento 01.

Nomenclatura	Qtd	Total
malignos		
Worm.Win32	2.382	
IM-Worm.Win32	244	
Net-Worm.Win32	1.580	4.617
IRC-Worm.Win32	49	
P2P-Worm.Win32	362	
benignos		5.383
Total		10.000

A TAB. 6.3 mostra os resultados obtidos através do método proposto.

TAB. 6.3: Comparação dos classificadores para detecção de *Worms*.

Classificador	TVP	TFP	TP	ACC
Naive Bayes	69,7%	35,1%	78,7%	69,7%
SVM	89,8%	10,1%	89,9%	89,8%
J48	92,6%	8,0%	92,7%	**92,6%**
CART	92,3%	8,2%	92,4%	92,3%
Random Forest	93,6%	6,8%	93,7%	**93,6%**

Nesse experimento, os algoritmos de melhor desempenho foram *Random Forest* e J48 com 93,6% e 92,6% de acurácia, respectivamente.

6.2 EXPERIMENTO 02

Nesse experimento, 20.034 arquivos (11.115 *malwares* e 8.919 não *malwares*) foram submetidos à análise conforme TAB. 6.4.

TAB. 6.4: Distribuição dos artefatos do experimento 02.

Nomenclatura	Qtd	Total
malignos		
Trojan-Banker	933	
Trojan-Clicker	790	
Trojan-DDoS	90	
Trojan-Downloader	2.383	11.115
Trojan-Dropper	2.059	
Trojan-IM	131	
Trojan-Mailfinder	217	
Trojan-Proxy	992	
Trojan-Ransom	30	
Trojan-Spy	3.490	
benignos		8.919
Total		20.034

A TAB. 6.5 mostra os resultados obtidos através do método proposto.

TAB. 6.5: Comparação dos classificadores para detecção de *Trojans*.

Classificador	TVP	TFP	TP	ACC
Naive Bayes	52,4%	38,4%	73,6%	52,4%
SVM	90,1%	11,0%	90,3%	90,1%
J48	91,7%	8,7%	91,7%	**91,7%**
CART	91,5%	9,0%	91,5%	91,5%
Random Forest	92,9%	7,4%	92,9%	**92,9%**

Nesse experimento, os algoritmos de melhor desempenho foram *Random Forest* e J48 com 92,9% e 91,7% de acurácia, respectivamente.

6.3 EXPERIMENTO 03

Nesse experimento, 18.510 arquivos (9.591 *malwares* e 8.919 não *malwares*) foram submetidos à análise conforme TAB. 6.6.

TAB. 6.6: Distribuição dos artefatos do experimento 03.

Nomenclatura	Qtd
Backdoor.Win32	9.591
benignos	8.919
Total	18.510

A TAB. 6.7 mostra os resultados obtidos através do método proposto.

TAB. 6.7: Comparação dos classificadores para detecção de *Backdoors*.

Classificador	TVP	TFP	TP	ACC
Naive Bayes	50,1%	46,5%	67,2%	50,1%
SVM	89,3%	11,0%	89,5%	89,3%
J48	91,6%	8,5%	91,6%	**91,6%**
CART	91,6%	8,5%	91,6%	91,6%
Random Forest	92,7%	7,4%	92,7%	**92,7%**

Nesse experimento, os algoritmos de melhor desempenho foram *Random Forest* e J48 com 92,7% e 91,6% de acurácia, respectivamente.

6.4 EXPERIMENTO 04

Nesse experimento, 17.919 arquivos (9.000 *malwares* e 8.9191 não *malwares*) foram submetidos à análise, conforme TAB. 6.8.

TAB. 6.8: Distribuição dos artefatos do experimento 04.

Nomenclatura	Qtd	Total
malignos		
Worm.Win32	3.000	
Trojan.Win32	3.000	9.000
Backdoor.Win32	3.000	
benignos		8.919
Total		17.919

A TAB. 6.9 mostra os resultados obtidos através do método proposto.

TAB. 6.9: Comparação dos classificadores para detecção de *Malwares*.

Classificador	TVP	TFP	TP	ACC
Naive Bayes	44,3%	8,9%	83,4%	44,3%
SVM	89,9%	10,2%	90,9%	89,9%
J48	90,4%	9,7%	90,8%	**90,4%**
CART	90,3%	9,7%	90,7%	90,3%
Random Forest	91,1%	9,0%	91,5%	**91,1%**

Nesse experimento, os algoritmos de melhor desempenho foram *Random Forest* e J48 com 91,1% e 90,4% de acurácia, respectivamente.

6.5 CONCLUSÃO PARCIAL

Em todos os experimetos os algoritmos foram executados na modalidade de validação cruzada, a fim de obter resultados estatisticamente mais significativos, utilizando-se 10 iterações. Podemos verificar que os algoritmos J48 e *Random Forest* apresentaram os melhores resultados, pois obtiveram a taxa de acurária maior que os demais algoritmos.

Para esses experimentos foi utilizado o *software* Weka 3.7, mas o mesmo apresentou dificuldade de processamento e mensagens de falta de memória com dados maiores, como é o caso do experimento 02. Como temos um acervo de mais de 360.000 *malwares* e a ideia é usar uma grande parte desse acervo para criar um classificador com a maior taxa de acurácia possível em uma plataforma de alto desempenho, o weka mostrou-se limitado para a tarefa. Além do fato de que para uma base de dados ser carregada no software é necessário que o arquivo esteja no formato .arff (*Attribute-Relation File Format*), formato de leitura para o Weka. A nossa base de dados está no formato .csv (*Comma-Separated Values*) e é necessário um *framework* que aceite esse formato ou, ao menos, deixe adicionar facilmente esse tipo de arquivo ao seu projeto.

Para os experimentos iniciais, o WEKA atendeu muito bem, mas para aplicação da proposta foi implementado uma solução própria utilizando o *framework* FAMA.

Com base no bom desempenho do J48 e do *Random Forest*, foi decidido implementar um algoritmo de árvore de decisão no FAMA. O algoritmo escolhido foi o ID3 que é o antecessor do C4.5, este último chamado de J48 no WEKA. Para alimentar o ID3 e o mesmo ter um desempenho igual ao J48, os dados foram passados para log base 2, pois os dados numéricos eram muito dispersos. Depois do algoritmo implementado e a base de dados submetida, foi obtida a taxa de acurácia esperada, como pode ser visto na FIG. 6.1. A partir do código do ID3, foi implemetado o *Random Forest*, que também alcançou desempenho esperado.

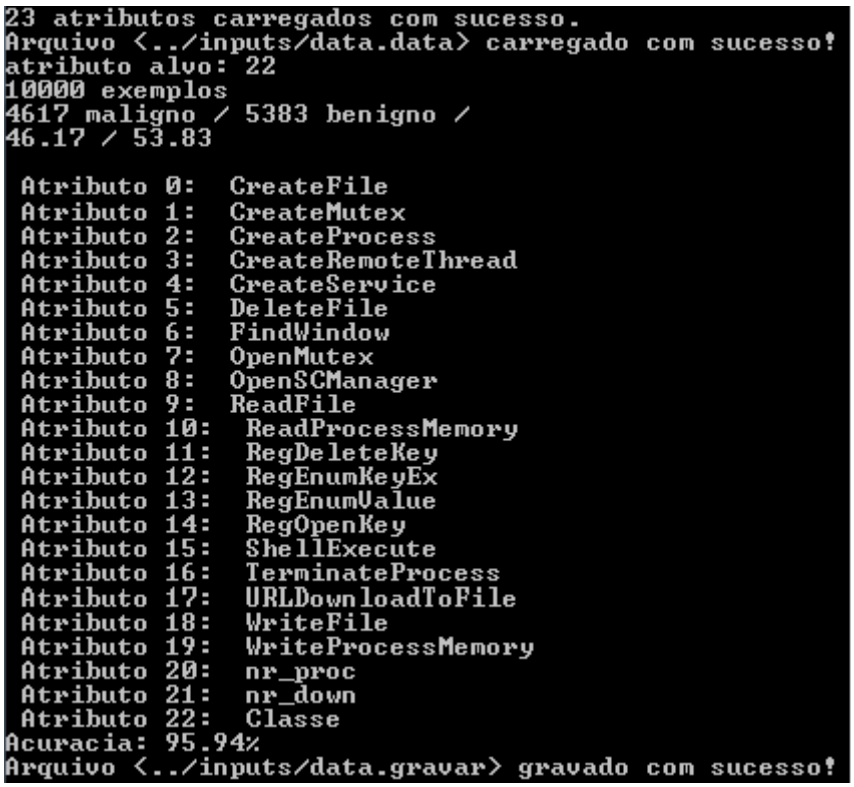

FIG. 6.1: Resultado do ID3 no FAMA.

7 CONSIDERAÇÕES FINAIS

7.1 CONCLUSÃO

Os *malwares* representam um desafio significativo para os pesquisadores. Além da grande velocidade de disseminação e criação de novas variantes, a sua natureza furtiva requer que pesquisadores desenvolvam técnicas efetiva para sua detecção. Para auxiliar esse esforço, este trabalho apresentou uma abordagem diferenciada das já disponíveis na literatura relacionada. Ao contrário dos métodos tradicionais, que incluem a análise manual para a geração de assinaturas, o foco da solução proposta é a análise automática utilizando sandbox e aprendizado de máquina. Para isso, foi proposta uma solução na qual tínhamos traçados alguns objetivos a serem alcançados como a instalação e configuração de um ambiente seguro e controlado para execução dos artefatos, que foi o *Cuckoo sandbox*; a criação de um mecanismo que permitisse a submissão automática para a análise de milhares de artefatos, que foi o *script* de submissão; tratamento dos relatórios produzidos pelo *sandbox*, que foi o *script* de cruzamento de funcionalidades; seleção do algoritmo de melhor desempenho, tarefa feita com o auxilio da ferramenta Weka; e por último, implementar o algoritmo de melhor desempenho, onde implementamos o ID3 e o *Random Forest* no *framework* FAMA. Desta maneira, todos os objetivos principais e secundários foram cumpridos.

A metodologia proposta neste trabalho tem por objetivo criar uma técnica que permita a análise de *malware* de forma automática e com um grau satisfatório de precisão. Nos experimentos de detecção de *malware*, foi analisado uma grande quantidade de artefatos e foi obtida taxa de acurácia acima de 90%. Apesar de não tratar as técnicas de anti-virtualização e *sleeping*, a proposta é totalmente viável para implementação no mundo real.

Com o objetivo de melhorar o desempenho da metodologia é possível selecionar e criar novos atributos, aumentar o grau de automatização da submissão e análise dos artefatos, utilizar outros algoritmos de aprendizado de máquina e comparar os resultados, além de treinar e testar o sistema com mais programas benignos cujo o comportamento se aproxime do comportamento de *malwares*.

7.2 TRABALHOS FUTUROS

Para trabalhos futuros, sugere-se encorporar, à essa proposta, mecanismos de detecção de emprego de técnicas de anti-virtualização e de *sleeping*, assim como também construir um *sandbox* que possa ser imune as técnicas anti-análise. Também, para o futuro, sugere-se aplicar essa proposta a outros SO e aos dispositivos móveis, como *tablets* e *smartphones*. É interessante o estudo de outros *sandboxes* e outros algoritmos de aprendizado de máquina. Também é interessante a classificação de *malwares* dentro de classes pré-definidas e rotuladas por um antivírus.

8 REFERÊNCIAS BIBLIOGRÁFICAS

APPLEGATE, S. D. Cybermilitias and political hackers: Use of irregular forces in cyberwarfare. *IEEE Security and Privacy*, 9:16–22, 2011. ISSN 1540-7993.

AVTEST. Avtest :: The independent it-security institute, Dezembro 2011. URL `http://www.av-test.org/en/`. http://www.av-test.org/en/.

BAECHER, P., KOETTER, M., DORNSEIF, M. e FREILING, F. The nepenthes platform: An efficient approach to collect malware. Em *In Proceedings of the 9 th International Symposium on Recent Advances in Intrusion Detection (RAID*, págs. 165–184. Springer, 2006.

BARFORD, P. e BLODGETT, M. Toward botnet mesocosms. Em *Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, págs. 6–6, Berkeley, CA, USA, 2007. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1323128.1323134`.

BAYER, U., KRUEGEL, C. e KIRDA, E. Ttanalyze: A tool for analyzing malware, 2006a.

BAYER, U., MOSER, A., KRUEGEL, C. e KIRDA, E. Dynamic analysis of malicious code. *Journal in Computer Virology*, 2:67–77, 2006b. ISSN 1772-9890. URL `http://dx.doi.org/10.1007/s11416-006-0012-2`.

BRAND, M., VALLI, C. e WOODWARD, A. Malware forensics: Discovery of the intent of deception. http://ro.ecu.edu.au/adf/75/, Novembro 2010. Security Research Centre Conferences, 8th Australian Digital Forensics Conference.

BREIMAN, L. Random forests. *Machine Learning*, 45:5–32, 2001. ISSN 0885-6125. URL

BREIMAN, L., FRIEDMAN, J. H., OLSHEN, R. A. e STONE, C. J. *Classification and Regression Trees*. Wadsworth International Group, Belmont, California, 1984.

BROWN, M., LEWIS, H. e GUNN, S. Linear spectral mixture models and support vector machines for remote sensing. *IEEE Transactions on Geoscience and Remote Sensing*, 38:2346–2360, 2000.

CAVALCA, D. e GOLDONI, E. An open architecture for distributed malware collection and analysis. Em HUEBNER, E. E ZANERO, S., editores, *Open Source Software for Digital Forensics*, págs. 101–116. Springer US, 2010. ISBN 978-1-4419-5802-0. URL `http://dx.doi.org/10.1007/978-1-4419-5803-7`$_7$.

CERT.BR. Cartilha de segurança para internet, Novembro 2012. URL `http://cartilha.cert.br/`. http://cartilha.cert.br/.

CHAKRABARTI, S. *Mining the Web: Discovering Knowledge from Hypertext Data*. Morgan-Kauffman, 2002. ISBN ISBN 1-55860-754-4. URL http://www.cse.iitb.ac.in/ soumen/mining-the-web/.

CHANG, C.-C. e LIN, C.-J. Libsvm: A library for support vector machines. *ACM Trans. Intell. Syst. Technol.*, 2(3):27:1–27:27, maio 2011. ISSN 2157-6904. URL http://doi.acm.org/10.1145/1961189.1961199.

CHEN, T. M. e ABU-NIMEH, S. Lessons from stuxnet. *Computer*, 44:91–93, 2011. ISSN 0018-9162.

CHRISTODORESCU, M. e JHA, S. Static analysis of executables to detect malicious patterns. Em *In Proceedings of the 12th USENIX Security Symposium*, págs. 169–186, 2003.

CHRISTODORESCU, M. e JHA, S. Testing malware detectors. Em *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '04, págs. 34–44, New York, NY, USA, 2004. ACM. ISBN 1-58113-820-2. URL http://doi.acm.org/10.1145/1007512.1007518.

COHEN, F. Computer viruses: theory and experiments. *Comput. Secur.*, 6(1):22–35, fevereiro 1987. ISSN 0167-4048. URL http://dx.doi.org/10.1016/0167-4048(87)90122-2.

COLLBERG, C. e NAGRA, J. *Surreptitious software: obfuscation, watermarking, and tamperproofing for software protection*. Addison-Wesley Professional, 2009.

DA FONSECA, J. M. M. R. Indução de Árvores de decisão. Dissertação de Mestrado, Universidade Nova de Lisboa, 1994.

DAVIS, M., BODMER, S. e LEMASTERS, A. *HACKING EXPOSED MALWARE AND ROOTKITS*. HACKING EXPOSED. 1th edition, setembro 2009.

EXÉRCITO. Boletim do exército nº 31/2010, Agosto 2010.

FACELI, K., LORENA, A. C., GAMA, J. e CARVALHO, A. *Inteligência Artificial - Uma Abordagem de Aprendizado de Máquina*. 2011.

FALLIERE, N., MURCHU, L. O. e CHIEN, E. W32.Stuxnet Dossier. Technical report, Symantic Security Response, outubro 2010.

FAMA. Fama, dezembro 2011. URL https://code.google.com/p/fama/.

FREILING, F. C., HOZ, T. e WICHERESKI, G. Botnet tracking: Exploring a root-cause methodology to prevent distributed denial-of-service attacks. Em SABRINA, SYVERSON, P. E GOLLMANN, D., editores, *European Symposium on Research in Computer Security*, volume 3679 of *Lecture Notes in Computer Science*. Springer, setembro 2005.

GOSTEV, A. Back to stuxnet: the missing link - securelist. http://www.securelist.com/en/blog/208193568/back_to_stuxnet_the_missing_link, 2012. URL http://www.securelist.com/en/blog/208193568/back$_t o_s tuxnet_t he_m issing_l ink$.

GREAMO, C. e GHOSH, A. Sandboxing and virtualization: Modern tools for combating malware. *IEEE Security and Privacy*, 9:79–82, 2011. ISSN 1540-7993.

HALL, M., FRANK, E., HOLMES, G., PFAHRINGER, B., REUTEMANN, P. e WITTEN, I. H. The weka data mining software: an update. *SIGKDD Explorations*, 11(1): 10–18, 2009.

HONEYNET. The honeynet project, dezembro 2012. URL http://www.honeynet.org/.

JOHN, G. e LANGLEY, P. Estimating continuous distributions in bayesian classifiers. Em *In Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, págs. 338–345. Morgan Kaufmann, 1995.

KANG, M. G., POOSANKAM, P. e YIN, H. Renovo: a hidden code extractor for packed executables. Em *Proceedings of the 2007 ACM workshop on Recurring malcode*, WORM '07, págs. 46–53, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-886-2. URL http://doi.acm.org/10.1145/1314389.1314399.

KANG, M. G., YIN, H., HANNA, S., MCCAMANT, S. e SONG, D. Emulating emulation-resistant malware. Em *Proceedings of the 1st ACM workshop on Virtual machine security*, VMSec '09, págs. 11–22, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-780-6. URL http://doi.acm.org/10.1145/1655148.1655151.

KASPERSKY. Kaspersky lab projects. securelist. kaspersky lab projects, maio 2012. URL http://www.securelist.com/en/threats/detect?chapter=125.

KHAN, H., MIRZA, F. e KHAYAM, S. A. Determining malicious executable distinguishing attributes and low-complexity detection. *J. Comput. Virol.*, 7(2):95–105, maio 2011. ISSN 1772-9890. URL http://dx.doi.org/10.1007/s11416-010-0140-6.

KOLTER, J. Z. e MALOOF, M. A. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research*, 7:2006, 2006.

KOLTER, J. Z. Learning to detect malicious executables in the wild, 2004.

KRUEGEL, C., KIRDA, E., MUTZ, D., ROBERTSON, W. e VIGNA, G. Automating mimicry attacks using static binary analysis. Em *In USENIX Security Symposium*, 2005.

LEDER, F. e WERNER, T. Know Your Enemy: Containing Conficker, To Tame a Malware. Technical report, The Honeynet Project, http://honeynet.org, abril 2009.

LI, H.-J., TIEN, C.-W., TIEN, C.-W., LIN, C.-H., LEE, H.-M. e JENG, A. B. Aos: An optimized sandbox method used in behavior-based malware detection. Em *ICMLC*, págs. 404–409, 2011.

LINN, C. e DEBRAY, S. Obfuscation of executable code to improve resistance to static disassembly. Em *Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03, págs. 290–299, New York, NY, USA, 2003. ACM. ISBN 1-58113-738-9. URL http://doi.acm.org/10.1145/948109.948149.

LISTON, T. e SKOUDIS, E. On the cutting edge: Thwarting virtual machine detection, 2006.

MARTIGNONI, L., CHRISTODORESCU, M. e JHA, S. Omniunpack: Fast, generic, and safe unpacking of malware. *Computer Security Applications Conference, Annual*, 0: 431–441, 2007. ISSN 1063-9527.

MCCALLUM, A. e NIGAM, K. A comparison of event models for naive bayes text classification, 1998.

MELGANI, F. e BRUZZONE, L. Classification of Hyperspectral Remote Sensing Images With Support Vector Machines. *IEEE Transactions on Geoscience and Remote Sensing*, 42:1778–1790, agosto 2004.

MITCHELL, T. M. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997. ISBN 0070428077, 9780070428072.

MONARD, M. e BARANAUSKAS, J. Conceitos sobre aprendizado de maquina. *Sistemas Inteligentes-Fundamentos e Aplicações*, págs. 89–114, 2003.

MOSER, A., KRUEGEL, C. e KIRDA, E. Limits of static analysis for malware detection. *Computer Security Applications Conference, Annual*, 0:421–430, 2007. ISSN 1063-9527.

O'KANE, P., SEZER, S. e MCLAUGHLIN, K. Obfuscation: The hidden malware. *IEEE Security and Privacy*, 9:41–47, 2011. ISSN 1540-7993.

PIETREK, M. Peering inside the pe: A tour of the win32 portable executable file format, março 1994. URL http://msdn.microsoft.com/en-us/magazine/ms809762.aspx.

POUGET, F., DACIER, M. e PHAM, V. H. Leurre.com: on the advantages of deploying a large scale distributed honeypot platform. Em *ECCE 2005, E-Crime and Computer Conference, 29-30th March 2005, Monaco*, MONACO, 03 2005. URL http://www.eurecom.fr/publication/1558.

PREDA, M. D., CHRISTODORESCU, M., JHA, S. e DEBRAY, S. A semantics-based approach to malware detection. *SIGPLAN Not.*, 42(1):377–388, janeiro 2007. ISSN 0362-1340. URL http://doi.acm.org/10.1145/1190215.1190270.

PRESIDÊNCIA. Decreto nº 6.703, DEZEMBRO 2008.

QUINLAN, J. R. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, março 1986. ISSN 0885-6125. URL http://dx.doi.org/10.1023/A:1022643204877.

QUINLAN, J. R. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993. ISBN 1-55860-238-0.

QUINLAN, J. R. http://www.rulequest.com/products.html, dezembro 2012. URL http://www.rulequest.com/products.html.

QUIST, D. e SMITH, V. Detecting the presence of virtual machines using the local data table. *Offensive Computing*, março 2006.

RAFFETSEDER, T., KRUEGEL, C. e KIRDA, E. Detecting system emulators. Em *Proceedings of the 10th international conference on Information Security*, ISC'07, págs. 1–18, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-75495-4, 978-3-540-75495-4. URL http://dl.acm.org/citation.cfm?id=2396231.2396233.

RAMAN, K. Selecting features to classify malware. Em 2012, I. S., editor, *InfoSec Southwest 2012*, volume InfoSec Southwest 2012. Adobe Systems Incorporated, InfoSec Southwest 2012, 2012. URL http://2012.infosecsouthwest.com/files/speaker$_m aterials/ISSW2012_S electing_F eatures_t o_C la$

RIECK, K., HOLZ, T., WILLEMS, C., D SSEL, P. e LASKOV, P. Learning and classification of malware behavior. Em ZAMBONI, D., editor, *DIMVA*, volume 5137 of *Lecture Notes in Computer Science*, págs. 108–125. Springer, 2008. ISBN 978-3-540-70541-3. URL http://dblp.uni-trier.de/db/conf/dimva/dimva2008.htmlRieckHWDL08.

RIECK, K., TRINIUS, P., WILLEMS, C. e HOLZ, T. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, págs. 639–668, 2011.

ROYAL, P., HALPIN, M., DAGON, D., EDMONDS, R. e LEE, W. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. Em *Proceedings of the 22nd Annual Computer Security Applications Conference*, ACSAC '06, págs. 289–300, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2716-7. URL http://dx.doi.org/10.1109/ACSAC.2006.38.

SÁ LUCAS, L. Árvore, florestas e sua função como preditores: uma aplicação na avaliação do grau de maturidade de empresas. *PMKT Revista Brasileira de Pesquisas de Marketing, Opinião e Mídia*, (6), Março 2011.

SCHULTZ, M. G., ESKIN, E., ZADOK, E. e STOLFO, S. J. Data mining methods for detection of new malicious executables. Em *IEEE Symposium on Security and Privacy'01*, págs. 38–49, 2001.

SHAFIQ, M., TABISH, S., MIRZA, F. e FAROOQ, M. Pe-miner: Mining structural information to detect malicious executables in realtime. Em KIRDA, E., JHA, S. E BALZAROTTI, D., editores, *Recent Advances in Intrusion Detection*, volume 5758 of *Lecture Notes in Computer Science*, págs. 121–141. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-04341-3. URL http://dx.doi.org/10.1007/978-3-642-04342-0$_7$.

SHARIF, M., YEGNESWARAN, V., SAIDI, H., PORRAS, P. e LEE, W. Eureka: A framework for enabling static malware analysis. Em *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS '08, págs. 481–500, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-88312-8. URL http://dx.doi.org/10.1007/978-3-540-88313-5$_3$1.

SHINOTSUKA, H. Malware authors using new techniques to evade automated threat analysis systems, outubro 2012. URL http://www.symantec.com/connect/blogs/malware-authors-using-new-techniques-evade-aut

SIDDIQUI, M., WANG, M. e LEE, J. Detecting trojans using data mining techniques. Em HUSSAIN, D., RAJPUT, A., CHOWDHRY, B. E GEE, Q., editores, *Wireless Networks, Information Processing and Systems*, volume 20 of *Communications in Computer and Information Science*, págs. 400–411. Springer Berlin Heidelberg, 2009. ISBN 978-3-540-89852-8. URL http://dx.doi.org/10.1007/978-3-540-89853-5$_4$3.

SIKORSKI e HONIG. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. 2012.

SINGHAL, P. e RAUL, N. Malware detection module using machine learning algorithms to assist in centralized security in enterprise networks. *CoRR*, abs/1205.3062, 2012. URL http://dblp.uni-trier.de/db/journals/corr/corr1205.htmlabs-1205-3062.

SZOR, P. *The art of computer virus research and defense*. Addison-Wesley, Upper Saddle River, NJ [u.a.], 2005. ISBN 0-321-30454-3.

TRINIUS, P., WILLEMS, C., HOLZ, T. e RIECK, K. A malware instruction set for behavior-based analysis. Em FREILING, F. C., editor, *Sicherheit*, volume 170 of *LNI*, págs. 205–216. GI, 2010. ISBN 978-3-88579-264-2. URL http://dblp.uni-trier.de/db/conf/sicherheit/sicherheit2010.htmlTriniusWHR10.

VAPNIC, V. The nature of statistical learning theory. 1995.

VIRUSLAB. viruslab, dezembro 2012. URL http://www.viruslab.com.br/.

VIRUSTOTAL. Virustotal - free online virus and malware scan., Outubro 2011. URL http://www.virustotal.com. http://www.virustotal.com.

VXHEAVENS. Vx.netlux, abril 2012. URL http://vx.netlux.org/index.html.

WANG, J.-H., DENG, P. S., FAN, Y.-S., JAW, L.-J. e LIU, Y.-C. Virus detection using data mining techniques. Em *International Carnahan Conference on Security Technology*, 2003.

WEKA. Weka, dezembro 1993. URL http://www.cs.waikato.ac.nz/ml/weka/.

WIKIPEDIA. Thomas bayes, Dezembro 2012. URL http://en.wikipedia.org/wiki/Thomas$_B$*ayes*.

WILLEMS, C., HOLZ, T. e FREILING, F. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security and Privacy*, 5(2):32–39, março 2007. ISSN 1540-7993. URL http://dx.doi.org/10.1109/MSP.2007.45.

WITTEN, I. H., FRANK, E. e HALL, M. A. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, Burlington, MA, 3 edition, 2011.

YE, Y., WANG, D., LI, T. e YE, D. Imds: intelligent malware detection system. Em *KDD*, págs. 1043–1047, 2007.

YOU, I. e YIM, K. Malware obfuscation techniques: A brief survey. *Broadband, Wireless Computing, Communication and Applications, International Conference on*, 0:297–300, 2010.

ZHANG, H. The optimality of naive bayes. Em *FLAIRS Conference*, 2004.

ZOLKIPLI, M. F. e JANTAN, A. Malware behavior analysis: Learning and understanding current malware threats. *Network Applications, Protocols and Services, International Conference on*, 0:218–221, 2010.