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

REINALDO JOSÉ MANGIALARDO

INTEGRANDO AS ANÁLISES ESTÁTICA E DINÂMICA NA IDENTIFICAÇÃO DE MALWARES UTILIZANDO APRENDIZADO DE MÁQUINA

Rio de Janeiro
2015

INSTITUTO MILITAR DE ENGENHARIA

REINALDO JOSÉ MANGIALARDO

INTEGRANDO AS ANÁLISES ESTÁTICA E DINÂMICA NA IDENTIFICAÇÃO DE MALWARES UTILIZANDO 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: Julio Cesar Duarte - D.Sc

Rio de Janeiro
2015

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

XXXXXMangialardo, R. J.
Integrando as análises estática e dinâmica na identificação de malwares utilizando aprendizado de máquina/ Reinaldo José Mangialardo.
– Rio de Janeiro: Instituto Militar de Engenharia, 2015.
xxx p.: il., tab.

Dissertação (mestrado) – Instituto Militar de Engenharia – Rio de Janeiro, 2015.

1. Análise de malwares. 2. Aprendizado de máquina. I. Título. II. Instituto Militar de Engenharia.

CDD 629.892

INSTITUTO MILITAR DE ENGENHARIA

REINALDO JOSÉ MANGIALARDO

INTEGRANDO AS ANÁLISES ESTÁTICA E DINÂMICA NA IDENTIFICAÇÃO DE MALWARES UTILIZANDO 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: Julio Cesar Duarte - D.Sc

Aprovada em xx de xxxxxx de xxxx pela seguinte Banca Examinadora:

Julio Cesar Duarte - D.Sc do IME - Presidente

Prof. Ronaldo Ribeiro Goldschmidt - D.Sc, do IME

Prof. Cicero Nogueira Dos Santos - D.Sc, da IBM

Rio de Janeiro
2015

Dedico este trabalho à minha esposa com muito amor, carinho e gratidão, pelo seu companheirismo, respeito, incentivo e paciência.

AGRADECIMENTOS

Agradeço a todas as pessoas que contribuíram com o desenvolvimento desta dissertação de mestrado, tenha sido por meio de críticas, idéias, apoio, incentivo ou qualquer outra forma de auxílio. Em especial, desejo agradecer às pessoas citadas a seguir.

Ao meu pai José (**in memoriam**) e minha mãe Rosemari pelos valores transmitidos, pelo apoio e carinho.

À minha esposa Andréa e aos meus queridos filhos Júlio e Yasmin pelo amor fraternal e familiar, solidariedade e companheirismo.

Ao meu orientador Maj Julio Cesar Duarte pela paciência, compreensão e orientação.

Por fim, a todos os professores e funcionários do Departamento de Engenharia de Sistemas (SE/8) do Instituto Militar de Engenharia.

Reinaldo José Mangialardo

Não devemos admitir mais causas para as coisas naturais do que as que são verdadeiras e suficientes para explicar suas aparências.

Isaac Newton

SUMÁRIO

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SÍMBOLOS

ABREVIATURAS

API	-	*Application Programming Interface*
ARPANET	-	*Advanced Research Projects Agency Network*
ASCII	-	*American Standard Code for Information Interchange*
CDCiber	-	*Centro de Defesa Cibernética*
CERT	-	*Centro de Estudos, Resposta e Tratamento de Incidentes de Segurança no Brasil*
COFF	-	*Common Object File Format*
CRC	-	*Cyclic Redundancy Check*
CTIR	-	*Centro de Tratamento de Incidentes de Segurança de Rede de Computadores da Administração Pública Federal*
DLL	-	*Dynamic Link Library*
EDA	-	*Electronic Design Automation*
ELF	-	*Extensible Linking Format*
FAMA	-	*Framework de Aprendizado de Máquina*
GIS	-	*Geographic Information System*
IBK	-	*Instance Based K-Nearest-Neighboors*
IME	-	*Instituto Militar de Engenharia*
IP	-	*Internet Protocol*
LabDCiber	-	*Laboratório de Defesa Cibernética*
MS-DOS	-	*Microsoft Disk Operating System*
PCAP	-	*Packet Capture*
SVM	-	*Support Vector Machines*
URL	-	*Uniform Resource Locator*
WEKA	-	*Waikato Environment for Knowledge Analysis*
Win PE 32	-	*Windows Portable Executable Format*

RESUMO

Sistemas de Análise e Classificação de *malwares* utilizam técnicas de análise estática e dinâmica, juntamente com algoritmos de aprendizado de máquina, para automatizar a tarefa de identificação e classificação de códigos maliciosos. Ambas as técnicas de análise possuem pontos fracos que permitem o emprego de técnicas de evasão da análise, dificultando a identificação de *malwares*.

Neste trabalho, propomos a unificação das análises estática e dinâmica de *malwares*, como um método que permita obter as características dos *malwares*, diminuindo a oportunidade de sucesso das técnicas de evasão. A partir dos dados obtidos na fase de análise, usamos os algoritmos de aprendizado de máquina C5.0 e Random Forest, implementados no *framework* de aprendizado de máquina FAMA, para executar a identificação e classificação dos *malwares* em duas classes, *malwares e não malwares* e também em múltiplas classes.

Mostramos que a acurácia da análise unificada obteve um resultado de 95,75% para a classificação dos códigos como *malwares e não malwares* e resultado igual a 93,02% para os *malwares* categorizados de acordo com seus tipos. Em todos os experimentos, a análise unificada produziu melhores resultados do que os obtidos pelas análises estática e dinâmica de *malwares* realizadas isoladamente.

ABSTRACT

Malware Analysis and Classification Systems use static and dynamic techniques, in conjunction with machine learning algorithms, to automate the task of identification and classification of malicious codes. Both techniques have weaknesses that allow the use of analysis evasion techniques, hampering the identification of malwares.

In this work, we propose the unification of static and dynamic analysis, as a method of collecting data from malware that decreases the chance of success for such evasion techniques. From the data collected in the analysis phase, we use the C5.0 and Random Forest machine learning algorithms, implemented inside the FAMA framework, to perform the identification and classification of malwares into two classes and multiple categories.

In our experiments, we showed that the accuracy of the unified analysis achieved an accuracy od 95.75% for the binary classification problem. and an accuracy value of 93.02% for the multiple categorization problem. In all experiments, the unified analysis produced better results than those obtained by static and dynamic analyzes isolated.

1 INTRODUÇÃO

Malwares (softwares maliciosos) são programas desenvolvidos para executar ações danosas em um computador. Os principais motivos que levam ao desenvolvimento e a propagação de códigos maliciosos são a obtenção de vantagens financeiras, a coleta de informações confidenciais, o desejo de autopromoção e o vandalismo. Além disto, os códigos maliciosos são muitas vezes usados como intermediários e possibilitam a prática de golpes, a realização de ataques e a disseminação de *spam* (CERT.BR, 2013). A detecção e análise de códigos maliciosos são atividades cruciais para qualquer mecanismo de defesa contra estes ataques. Para identificar os *malwares* existem duas técnicas: a análise estática de *malwares* e a análise dinâmica de *malwares*. A análise estática permite extrair características do código sem executá-lo e a análise dinâmica permite extrair informações quando o código é executado. Os desenvolvedores de códigos maliciosos utilizam técnicas antianálise ou de evasão das análises estática e dinâmica de *malwares* para evitar que informações sobre os *malwares* sejam obtidas, dificultando ou impedindo a sua identificação. Além deste problema, ainda existe a demanda de tempo para classificar o código como um *malware* pelo especialista de segurança. Esta tarefa não é simples e para que seja possível é necessário o planejamento e desenvolvimento de sistemas que executem a tarefa de forma automatizada, reduzindo o tempo de resposta de reação a ação dos *malwares*. Técnicas de aprendizado de máquina têm sido utilizadas em conjunto com algoritmos de coleta estática e dinâmica para a identificação e classificação de *malwares*, de tal forma a automatizar a classificação e reduzir o tempo de desenvolvimento de *softwares* de antivírus para combater os códigos maliciosos. Estas técnicas de aprendizado de máquina são capazes de aprender uma descrição generalizada de um *malware* e aplicar este conhecimento para classificar novos códigos maliciosos que ainda não são conhecidos.

1.1 MOTIVAÇÃO

O emprego de *malwares* tem sido um recurso amplamente utilizado para realizar ataques contra sistemas de informações pessoais, bancárias, empresariais e governamentais.

Malwares vêm sendo utilizados por países como instrumento de política externa e podemos citar como exemplos de empregos os ataques contra o programa nuclear irani-

ano, com a utilização de operações encobertas, que fizeram o uso de *malwares* como o Stuxnet e o Flame (DEARAÚJO JORGE, 2012). Para obter informações sobre os códigos maliciosos os especialistas de segurança utilizam as técnicas de análise estática e dinâmica de *malwares*. Porém, estas técnicas possuem pontos fracos que permitem ao atacante empregar técnicas de ofuscação de código (CHRISTODORESCU, 2006) para dificultar ou impedir a análise estática ou a detecção de máquina virtual (OKTAVIANTO, 2013) para impedir ou dificultar a análise dinâmica de *malwares*. Estas análises fornecem os dados que são utilizados no processo de classificação, predição e identificação de *malwares*, sendo importante para o desenvolvimento de *softwares* antivírus. Quando uma técnica de evasão é aplicada com sucesso, os resultados da análise ficam prejudicados devido à falta de uma informação sobre o *malware*. Com o objetivo de melhorar o desempenho da análise e diminuir o impacto das atividades de evasão, colaborando com a defesa cibernética, este trabalho propõe a integração ou unificação das análises estática e dinâmica de *malwares* e utilização de aprendizado de máquina para automatizar a classificação. Com a técnica de unificação, mesmo que alguma característica de uma das análises fique oculta pelas técnicas de evasão, outras ainda estarão presentes, permitindo a identificação e classificação dos *malwares*.

1.2 OBJETIVOS

Unificar as classificações estática e dinâmica de *malwares* e otimizar a capacidade de detecção e classificação de *softwares* maliciosos, utilizando algoritmos de aprendizado de máquina.

1.3 JUSTIFICATIVA

Os *malwares* podem ser utilizados, de forma direta ou indireta, como um instrumento para a execução de vários ataques contra sistemas de computadores conectados em rede, em especial à Internet. A estes ataques chamamos de ataques cibernéticos e ao conjunto de ações para contê-los chamamos de Defesa Cibernética.

A Defesa Cibernética é um "conjunto de ações defensivas, exploratórias e ofensivas, no contexto de um planejamento militar, realizadas no espaço cibernético, com as finalidades de proteger os sistemas de informação, obter dados para a produção do conhecimento de inteligência e causar prejuízos aos sistemas de informação do oponente" (LABDCIBER-

IME, 2014).

Para ilustrar a importância do assunto, mostramos na figura 1.1 as estatísticas publicadas pelo Centro de Tratamento de Incidentes de Segurança de Rede de Computadores da Administração Pública Federal - CTIR Gov referentes às notificações e incidentes de segurança no 2^o trimestre de 2014. O gráfico foi produzido em parceria com o Centro de Defesa Cibernética - CDCiber e com o Grupo de Resposta a Incidentes de Segurança para a Internet Brasileira - CERT.BR e nele é possível observar que houve um aumento da quantidade de notificações e de incidentes no período em que ocorreu a Copa do Mundo da FIFA Brasil 2014.

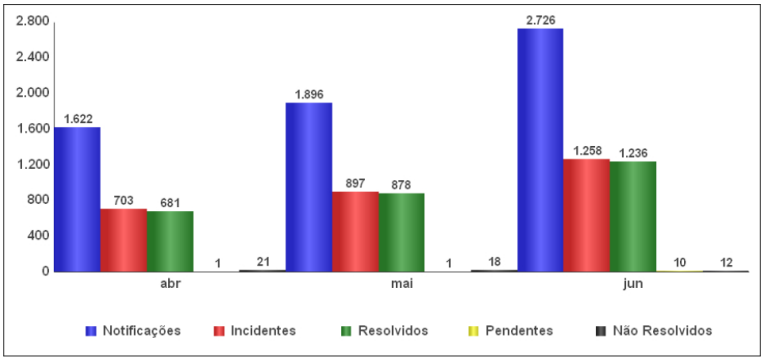


FIG. 1.1: Distribuição de notificações por status e mês de criação
Fonte: (CTIR, 2014)

Devido aos ataques cibernéticos, a Estratégia Nacional de Defesa, aprovada pelo Decreto nº 6.703, de 18 de dezembro de 2008, considera que um dos setores estratégicos da defesa é o cibernético e a defesa cibernética passou a ser uma prioridade para o Exército Brasileiro (LABDCIBER-IME, 2014).

Desta forma, uma pesquisa sobre a detecção e classificação de *malwares* de forma integrada e automatizada, utilizando algoritmos de aprendizado de máquina, poderá ser útil, melhorando a capacidade de identificação e classificação dos códigos maliciosos, contribuindo com a defesa cibernética.

1.4 ORGANIZAÇÃO DA DISSERTAÇÃO

Esta dissertação está organizada da seguinte forma: o capítulo 2 apresenta conceitos sobre características dos arquivos e seus formatos, *malwares*, técnicas de análise de *malwares* e técnicas de evasão de *malwares*; o capítulo 3 apresenta os algoritmos e *frameworks* de

aprendizado de máquina. No capítulo 4 são apresentados alguns trabalhos relacionados ao projeto de pesquisa desenvolvido. O capítulo 5 apresenta as etapas que foram seguidas para unificar as análises estática e dinâmica de *malwares*. O capítulo 6 apresenta e discute os resultados obtidos. No capítulo 7 é feita a conclusão e são dadas sugestões de trabalhos futuros.

2 ANÁLISE DE MALWARES

A análise de *malwares* é o processo de identificação de códigos maliciosos. Compreende atividades para a identificação de suas características baseando-se na forma como são obtidos, como são instalados, como se propagam, e como atuam nos sistemas. O *malware* é um *software* desenvolvido com o objetivo de causar danos a um sistema de computador que pode ou não estar conectado a uma rede de computadores. Pode ser utilizado pelo atacante para prejudicar o desempenho e/ou o funcionamento de um sistema e/ou prejudicar o funcionamento de uma rede de computadores. Sua motivação pode ser a obtenção de ganhos ilícitos, causar prejuízos a uma organização, a uma pessoa, ao funcionamento de infraestruturas críticas como usinas hidrelétricas, usinas nucleares, sistemas de transporte de cidades, etc. Podem também ser empregados com objetivos militares (guerra cibernética) e políticos (hacktivismo).

Os códigos maliciosos estão presentes em arquivos que são executados em computadores. Os arquivos possuem uma série de características que permitem identificar um *malware*. Como *malwares* são arquivos que são executados em sistemas de computadores, as propriedades dos formatos de arquivos devem ser conhecidas, permitindo que a natureza e o propósito do arquivo sejam avaliados.

2.1 OS FORMATOS DE ARQUIVOS

Um formato de arquivo define como *bits* são utilizados para armazenar uma informação em um meio digital.

Uma especificação de formato informa os detalhes necessários para construir um arquivo válido de um tipo específico. Permite que o sistema operacional do computador execute o arquivo ou que uma aplicação instalada no computador o manipule. Sem a especificação de formato, o arquivo torna-se somente uma sequência de zeros e uns sem significado algum para o sistema computacional. Formatos de arquivo podem ser dependentes ou independentes do *software*. Exemplos de formatos de arquivos independentes de *sofware* são os formatos ASCII e UNICODE. Exemplos de arquivos dependentes de *software* são os arquivos executáveis com formato Win PE32, dependente do sistema operacional Windows, e com o formato ELF usado em sistemas operacionais Unix, Solaris e

Linux (UFF, 2014).

Os formatos de arquivos podem ter especificações proprietárias e fechadas, proprietárias e abertas e não proprietárias e abertas.

As especificações proprietárias e fechadas estão sobre o domínio de um desenvolvedor que detém o código e que não disponibiliza o código e suas especificações; nas especificações proprietárias e abertas, a especificação é disponibilizada para que desenvolvedores produzam *softwares* que possam ler ou executar arquivos. Exemplo: formato pdf e formato Win PE; nas especificações não proprietárias e abertas, o código e as especificações são disponibilizados ao público. Exemplos: o *framework* de aprendizado de máquina FAMA (FAMA, 2014) e o *framework* de aprendizado de máquina Weka (WEKA, 2014) para solução de problemas de mineração de dados.

Existem vários formatos de arquivos (ODP, 2014) e alguns deles são apresentados na tabela 2.1. Os formatos estão agrupados em classes conforme o emprego em diversas plataformas.

Dentre estes formatos, o formato de arquivo Win PE32 foi utilizado como objeto de análise nesta pesquisa. Ele possui especificação aberta, possibilitando manter uma padronização da análise. Também, facilita investigar as características dos *malwares*, devido ao volume de exemplos conhecidos e utilizados para atacar o sistema operacional Windows e as aplicações que são executadas neste ambiente.

O formato Windows Portable Executable é uma especificação de formato de arquivo que descreve a estrutura de um arquivo executável (imagem) e objetos utilizados pelo sistema operacional Microsoft Windows. Deriva-se do formato Common Object File Format (COFF) e aplica-se aos códigos executáveis, às bibliotecas de vínculos dinâmicos (DLLs) e *drivers* do Kernel.

Uma característica importante do arquivo PE é que a sua estrutura quando armazenada na memória é praticamente a mesma que a definida para o armazenamento em disco.

A figura 2.1 ilustra a especificação do arquivo PE. Consiste em uma série de subestruturas e componentes que o descrevem e que contém *flags* de estado, ponteiros, dados e código. A estrutura, como se pode ver na figura 2.1 divide-se em: MS-DOS Header, MS-DOS Stub, PE Header, Data Directory e Section Table.

O MS-DOS Header é a estrutura de inicio de todo arquivo PE. Contém duas informações importantes: o campo *e_magic* que contém a assinatura do arquivo executável, inicialmente identificado pelos caracteres MZ ou pelos caracteres hexadecimais 4D 5A. O

TAB. 2.1: Exemplos de formatos de arquivos

CATEGORIA	FORMATO
ARCHIVE	Arquivo compactado pelo programa 7 zip.
	Gzip (.gz) Arquivo compactado pelo programa Gzip.
	JAR (Java Archive): usado para distribuir classes Java.
	RAR (Roshal Archive) formato fechado de compactação de arquivos.
ARCHIVING	ISO imagem de arquivos de um CD, DVD ou qualquer disco óptico.
CAD	DXF ASCII Drawing Interchange file format; AutoCAD.
	DWG AutoCAD and Open Design Alliance applications.
EDA	VCD; formato para simulação de ondas digitais.
Database	DB Paradox; SQLite.
	DBF db/dbase, Oracle.
	MDF Microsoft SQL Server Database.
	ODB LibreOffice Base ou OpenOffice Base database.
	ORA Oracle tablespace files.
Documento	CSV texto ASCII com valores separados por vírgula.
	DOC documento do Microsoft Word.
	DOCX documento do Office Open XML.
	HTML HyperText Markup Language. (.html, .htm)
	ODT documento OpenDocument text.
Font File	PostScript Font Type 1, Type 2.
	TTF (.ttf, .ttc) TrueType Font.
GIS	GML Geography Markup Language file.
	GPX XML based interchange format.
	NTF National Transfer Format file.
Graphics	BMP Microsoft Windows Bitmap formatted image.
	GIF CompuServe's Graphics Interchange Format.
	ICO formato de arquivos utilizado para ícones no MS Windows.
	JPEG (.jpg or .jpeg) Joint Photographic Experts Group
	PNG Portable Network Graphic.
	TIFF (.tif or .tiff) Tagged Image File Format.
Links e Shortcuts	LNK atalho para um arquivo binário (MS Windows).
	URL; INI arquivo ponteiro para uma URL
Executáveis e DLLs	APK Android Package
	COM files processador de comandos DOS
	ELF Executable and Linkable Format.
	DOS executable (.exe usado no DOS)
	Portable Executable (.EXE, .DLL).
Script	JS JavaScript e Jscript.
	PHP.

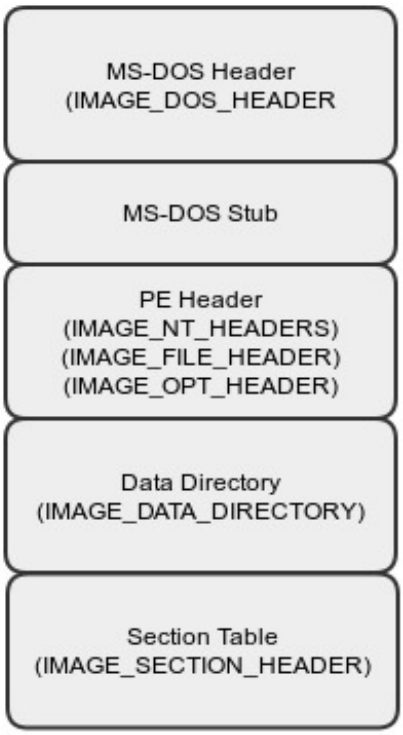


FIG. 2.1: Formato de Arquivo Windows Portable Executable
Fonte: (H.MALIN, 2012)

campo *e_lfanew* é um ponteiro com o *offset* que indica aonde o PE header inicia.

O MS-DOS *stub* contém um código de programa utilizado para identificar a compatibilidade da aplicação. Quando o formato PE foi criado, muitos usuários utilizavam o sistema MS-DOS e não o ambiente gráfico. Se um arquivo PE tentar executar no ambiente DOS, uma mensagem será impressa na tela do sistema pelo MS-DOS *stub*: *"This program cannot be run in DOS mode"*. O programa *stub* não é essencial para o funcionamento correto de um arquivo PE e muitas vezes os *hackers* irão modificá-lo, apagá-lo ou até mesmo ofuscá-lo.

Depois do MS-DOS *stub*, no endereço resultante do *offset* armazenado no campo *e_lfanew* encontra-se a estrutura PE Header. O PE Header é constituído pelas estruturas IMAGE_FILE_HEADER e IMAGE_OPTIONAL_HEADER. O arquivo PE é identificado por uma assinatura de 4 *bytes* (DWORD) seguida por dois caracteres nulos (valor hexadecimal ASCII 50 45 00 00). A primeira subestrutura da estrutura PE_Header é a IMAGE_FILE_HEADER. Esta estrutura contém várias informações importantes: hora e data que o arquivo foi compilado/criado; plataforma e processador compatíveis com o

arquivo; número de seções na estrutura Section Table; características do arquivo, como, por exemplo, se o arquivo é um executável.

A estrutura IMAGE_OPTIONAL_HEADER, apesar do nome, é obrigatória para que um arquivo executável funcione no sistema. Esta é uma estrutura bastante extensa e contém informações importantes como a versão do *link editor* utilizado pelo compilador para produzir o arquivo executável, as características das DLLs, ponteiro para o endereço do ponto de entrada do programa e a versão do sistema operacional.

A estrutura IMAGE_DATA_DIRECTORY, normalmente chamada de Data Directory, contém 16 subdiretórios que permitem identificar e mapear as localizações de outras estruturas e seções em um arquivo PE.

A última estrutura de um arquivo PE é a IMAGE_SECTION_HEADER ou Section Table. Consiste de entradas individuais, ou cabeçalhos de seções, cada uma delas com tamanho igual a 40 *bytes* e que contém o nome, tamanho e descrição da respectiva seção. A estrutura IMAGE_FILE_HEADER (COFF Header) contém o campo *"NumberOfSections"* que identifica o número de entradas na Section Table.

As seções mais comuns presentes em um arquivo PE são: *code section* ou a seção de código (.text ou .code), *resource section* ou a seção de recursos (.rsrc), *data section* ou a seção de dados (.data), a *export section* ou seção de exportação de dados (.edata), a *import data section* ou seção de importação de dados (.idata) e *debug information* ou a seção de informações de *debug*.

Na seção de código fica armazenado o código compilado do programa e uma alteração no código produz uma mudança nos dados armazenados na estrutura de dados da seção.

A seção de dados pode ser subdividida em três subseções: a subseção BSS contém as variáveis não inicializadas; RDATA contém dados de somente leitura; e, DATA contém todas as outras variáveis que não pertencem às seções BSS e RDATA.

A seção de recursos contém todos os outros tipos de dados utilizados por um executável. Exemplo: ícones, imagens, disposição dos objetos nas janelas, etc.

A seção de exportação contém informações dos nomes e endereços de funções contidas em uma DLL. Um aspecto importante que deve ser analisado na busca de evidência de um código malicioso é a forma como o arquivo foi compilado. Um arquivo no ambiente Windows pode ser executado de forma estática ou dinâmica. A forma como o executável foi compilado, como ele foi codificado, impactam o conteúdo e o tamanho do arquivo.

Um executável estático é compilado com todas as bibliotecas e códigos necessários à sua

execução. Assim, o programa é autocontido. Por outro lado, um executável dinâmico, vincula em tempo de execução várias bibliotecas que ele precisa para executar. Estas bibliotecas e códigos vinculados dinamicamente são chamados de dependências e consistem basicamente de DLLs.

Outras informações importantes podem ser encontradas na estrutura da Section Table como, por exemplo, as informações de endereço relativo virtual do início da sessão (VirtualAddress), o tamanho total da seção em disco (SizeOfRawData), considerando-se o alinhamento utilizado na compilação, a posição da seção dentro do arquivo armazenado no disco (PointerToRawData) e as características da seção, como, por exemplo, se ela é de escrita/leitura.

2.2 ANÁLISE DE MALWARES

2.2.1 CLASSIFICAÇÃO DOS MALWARES

Malwares (*malicious softwares* ou softwares maliciosos) apareceram pela primeira vez em 1949. O conceito de vírus de computador surgiu em um trabalho de Cohen (SZOR, 2005) que elaborou um modelo formal matemático para um vírus de computador baseando-se no modelo de autômato celular com capacidade de auto reprodução (NEUMANN, 1966). Na década de 1970, Robert Thomas Morris elaborou o primeiro vírus de computador, chamado Creeper, que tinha a capacidade de infectar máquinas IBM 360 na ARPANET. Para eliminar este programa foi desenvolvido um outro programa chamado Reaper, dando origem aos programas antivírus (SECURITY, 2014).

Com a evolução da informática e das redes de computadores, em especial da Internet, vários outros tipos de *malwares* surgiram, motivados principalmente pela possibilidade de obtenção de lucros ilegítimos na Internet.

Para identificar, classificar e eliminar os *malwares*, os especialistas observam suas características comportamentais, entendem estas características e desenvolvem as medidas defensivas. Definir e identificar estas características é uma tarefa difícil, porque existem diferentes classificações e diariamente surgem variantes que mesclam características dos demais códigos (CERT.BR, 2013).

Pode-se citar como exemplos de classificação as adotadas pelo CERT.BR e a classificação de SZOR (SZOR, 2005).

Szor classifica os *malwares* nas classes vírus, worm, trojan, backdoor, downloader,

dropper e rootkit e o CERT.BR acrescenta a esta classificação as seguintes classes: bot, spyware, keylogger, adware, trojan downloader, trojan dropper, trojan backdoor, trojan DoS, trojan destrutivo, trojan clicker, trojan proxy e trojan banker.

O (CERT.BR, 2013) descreve as classes de malwares da seguinte forma:

Vírus é um programa ou parte de um programa de computador, normalmente malicioso, que se propaga inserindo cópias de si mesmo e se tornando parte de outros programas e arquivos. Foi o primeiro tipo de *malware* classificado e (COHEN, 1994) o definiu informalmente como "*um programa que é capaz de infectar outros programas modificando-os para incluir uma possível cópia evoluída de si mesmo*".

Para que possa se tornar ativo e dar continuidade ao processo de infecção, o vírus depende da execução do programa ou arquivo hospedeiro, ou seja, para que o seu computador seja infectado é preciso que um programa já infectado seja executado. Os principais meios de propagação atuais de vírus são os *pen-drives* e anexos de *e-mails*.

Há diferentes tipos de vírus. Alguns procuram permanecer ocultos, infectando arquivos do disco e executando uma série de atividades sem o conhecimento do usuário. Há outros que permanecem inativos durante certos períodos, entrando em atividade apenas em datas específicas. São exemplos de tipos de vírus mais comuns: os vírus propagados por *e-mail*, os vírus de *script*, os vírus de macro e os vírus de telefones celulares e *smart fones*.

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

O processo de propagação e infecção dos *worms* compreende as seguintes etapas: identificação dos computadores alvos; envio das cópias e ativação das cópias.

O *worm* não se propaga por meio da inclusão de cópias de si mesmo em outros programas ou arquivos, mas sim pela execução direta de suas cópias ou pela exploração automática de vulnerabilidades existentes em programas instalados em computadores. Assim, ele não depende de outro programa para disseminar um ataque.

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

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

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

Botnet é uma rede formada por centenas ou milhares de computadores zumbis e que permite potencializar as ações danosas executadas pelos *bots*. Zumbis são computadores contaminados por *bots* e quanto mais zumbis participarem da *botnet* mais potente ela será.

São exemplos de ações maliciosas por *botnets*: ataques de negação de serviço, propagação de códigos maliciosos (inclusive do próprio *bot*), coleta de informações de um grande número de computadores, envio de *spam* e camuflagem da identidade do atacante (com o uso de *proxies* instalados nos zumbis).

O funcionamento básico de uma *botnet* compreende as seguintes etapas: um atacante propaga um tipo específico de *bot* na esperança de infectar e conseguir a maior quantidade possível de zumbis; os zumbis ficam então à disposição do atacante, agora seu controlador, à espera dos comandos a serem executados; quando o controlador deseja que uma ação seja realizada, ele envia aos zumbis os comandos a serem executados, usando, por exemplo, redes do tipo P2P ou servidores centralizados; os zumbis executam os comandos recebidos, durante o período predeterminado pelo controlador; quando a ação se encerra, os zumbis voltam a ficar à espera dos próximos comandos a serem executados.

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

Pode ser usado tanto de forma legítima quanto maliciosa, dependendo de como é instalado, das ações realizadas, do tipo de informação monitorada e do uso que é feito por quem recebe as informações coletadas.

Alguns tipos específicos de programas *spyware* são o *keylogger* que permite capturar e armazenar as teclas digitadas pelo usuário no teclado do computador, o *screenloger* que permite armazenar a posição do cursor e a tela apresentada no monitor e o *adware* muito utilizado para divulgar propagandas. O *adware* é considerado malicioso, quando as propagandas apresentadas são direcionadas, de acordo com a navegação do usuário e sem que este saiba que tal monitoramento está sendo feito.

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

backdoor ilegítimo, utiliza mecanismos que permitem perpassar a verificação de segurança do sistema.

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

Basendo-se na nomeação de *malwares* de programas antivírus, o (CERT.BR, 2013) define que os cavalos de tróia dividem-se nas seguintes classes:

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

Rootkit é um conjunto de programas e técnicas que permitem esconder e assegurar a presença de um invasor ou de outro código malicioso em um computador comprometido. Pode ser usado para: remover evidências em arquivos de *logs*, instalar outros códigos maliciosos, como *backdoors*, para assegurar o acesso futuro, esconder atividades e informações, como arquivos, diretórios, processos, chaves de registro, conexões de rede, etc, mapear potenciais vulnerabilidades em outros computadores, por meio de varreduras na rede e capturar informações da rede onde o computador comprometido está localizado, pela interceptação de tráfego.

O objetivo de um *rootkit* é a manutenção do privilégio de acesso adquirido e não a obtenção de acesso privilegiado.

Pode-se verificar pelas definições dos diferentes tipos de *malwares*, que cada tipo de código malicioso possui características próprias que o define e o diferencia dos demais tipos, como a forma de obtenção, forma de instalação, meios usados para propagação e ações maliciosas mais comuns executadas nos computadores infectados. O CERT.BR generaliza todos os Trojans em uma categoria única que chamamos de *trojan*. Desta forma, para a finalidade deste trabalho, utilizaremos a seguinte classificação para os tipos de *malwares: vírus, worm, bot, trojan, spyware, backdoor* e *rootkit*. Esta classificação não é definitiva, devido à falta de padronização, a grande variabilidade e a grande proliferação dos *malwares*.

2.2.2 TÉCNICAS DE ANÁLISE DE MALWARES

Para identificar *malwares*, especialistas de segurança adotam um método geral de investigação que se divide normalmente em 5 (cinco) fases (H.MALIN, 2012):

- Fase 1: preservação forense e exame dos dados voláteis;
- Fase 2: exame da memória do sistema infectado;
- Fase 3: análise forense: exame dos meios de armazenamento permanentes (disco rígido, fitas de backup, etc.);
- Fase 4: verificação das características do arquivo desconhecido;
- Fase 5: análise estática e dinâmica do arquivo suspeito.

Cada uma destas fases possuem suas metodologias e finalidades formalizadas e permitem que investigadores digitais reconstruam uma imagem nítida dos eventos, possibilitando a identificação e obtenção de um detalhado entendimento do *malware* e que medidas defensivas sejam efetivadas, como o desenvolvimento de programas antivírus. Todo este processo é bastante complexo e demanda tempo e conhecimento do analista. Assim, quando um *malware* novo surge, a contramedida de segurança para ele não estará pronta e os sistemas ficarão expostos à ação do código malicioso. Quando isto ocorre, chama-se o código malicioso de *zero-day malware* (GOYAL, 2012) pois as assinaturas dos *softwares* antivírus ainda não estão disponíveis.

A fase 1 do trabalho de análise compreende a preservação do estado dos dados voláteis que são aqueles que existem somente enquanto o sistema estiver ligado. Os valores dos dados voláteis não se limitam somente aos dados armazenados na memória principal e que estejam associados ao *malware* mas, também, refere-se a outras informações como, por exemplo, endereços IP, registros de eventos de segurança em *logs* do sistema, além de outros detalhes que juntos podem fornecer um completo entendimento do *malware* e seu uso. A fase 2 compreende a captura de *dumps* de memória. Permite obter informações sobre o *malware* e evidências de sua atuação. Para que seja possível, o especialista precisa ter um profundo conhecimento das estruturas de dados na memória. Necessita de *softwares* que auxiliem no processo de verificação devido ao grande volume de estruturas de dados presentes nos *dumps* de memória. Na fase 3, o especialista busca encontrar evidências da presença do *malware* em locais como o disco rígido, incluindo os arquivos armazenados, arquivos de registros e *logs*, datas associadas a eles, etc. Estas informações podem ser úteis para identificar, por exemplo, a fonte do ataque e a funcionalidade do *malware*. Na fase 4 ocorre a análise inicial do arquivo suspeito. Nesta fase busca-se entender para que serve o arquivo, como ele pode ser caracterizado, por exemplo, como benigno ou maligno, se o arquivo possui conteúdo ofuscado, se sua origem é desconhecida, se é um arquivo conhecido mas, que está armazenado em local pouco usual, se produz atividade de rede anormal. Esta fase emprega uma análise estática inicial do código binário para obter informações como dependências do arquivo, assinaturas de antivírus, metadados associados com o arquivo suspeito. Na fase 5 as duas técnicas de análise de *malwares*, a técnica de análise estática e a análise dinâmica de *malwares* são empregadas e informações importantes são obtidas, permitindo que o analista faça a classificação do arquivo suspeito. A análise estática compreende a análise do código binário do arquivo, sem executá-lo,

enquanto que, a análise dinâmica ou comportamental compreende a execução do código e o monitoramento de seu comportamento, interação, e efeitos no sistema infectado.

2.2.2.1 ANÁLISE ESTÁTICA DE MALWARES

A análise estática de *malwares* é o processo de análise do código ou da estrutura de um programa para determinar a sua função. A análise é feita sem que o arquivo seja executado no sistema. Subdivide-se em análise estática básica e análise estática avançada (SIKORSKI, 2012).

Na análise estática básica empregam-se as seguintes técnicas na identificação de *malwares*: utilização de programas antivírus, utilização de *hashes* para identificar o arquivo, captura de informações existentes em *strings*, funções e cabeçalhos do arquivo. Na análise estática avançada o código do programa é desmontado, utilizando-se um *desmontador*, e é feita uma engenharia reversa com o objetivo de descobrir o que o programa faz.

O programa antivírus, recurso utilizado na análise estática básica, é ainda a defesa mais utilizada contra os *malwares*, apesar de muitos dos seus inconvenientes. Em geral, os antivírus trabalham de duas maneiras para identificar *malwares*: assinaturas e/ou heusrísticas. 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. A principal desvantagem desta técnica de detecção é que não é possível gerar as assinaturas em tempo real para que não exista a possibilidade de atuação dos *malwares* nos sistemas. A detecção heurística é uma técnica utilizada na tentativa de evitar os efeitos da defasagem da criação de assinaturas de novos códigos maliciosos, disponibilizadas nos bancos de dados dos antivírus. Nesta técnica o antivírus, a partir de determinadas características e regras, identifica o arquivo como suspeito ou não. Exemplo: o programa antivírus verifica que um determinado programa enviou três arquivos idênticos em sequência para um determinado endereço de *e-mail*. Esta atividade é considerada como suspeita (poderia ser atividade de um vírus de *script*) e o programa é marcado como suspeito pelo antivírus. Esta técnica também possui desvantagens. Além de ser muito difícil determinar todas as regras heurísticas que identifiquem atividades suspeitas, também a técnica costuma gerar muitos falsos alarmes, os falso-positivos. Um falso-positivo ocorre quando o antivírus identifica incorretamente um arquivo como *malware* e um falso-negativo ocorre quando o antivírus não consegue

identificar o *malware*. Assim, percebe-se que um antivírus pode falhar deixando o sistema vulnerável. Para aumentar a certeza acerca da classificação do arquivo como *malware* ou não pode-se utilizar sistemas de verificação disponibilizados na Internet. Um exemplo destes sistemas é o VírusTotal. O VírusTotal (SISTEMAS, 2014) é um serviço gratuito, disponível na Web, que permite analisar arquivos e URLs suspeitas, facilitando a rápida detecção de vírus, worms, cavalos de tróia e todos os tipos de arquivos maliciosos. Atualmente, o VírusTotal permite verificar como é feita a análise por mais de 50 programas antivírus para um arquivo submetido ao sistema. Além do VírusTotal existem outros mecanismos na Web que disponibilizam serviços com propósito idêntico. A tabela 2.2 apresenta alguns dos serviços de identificação de *malwares* disponíveis na Web.

Algoritmos *hash* são utilizados para criar uma identificação digital única para arquivos. Permite verificar se um arquivo não foi corrompido, pela comparação dos *hashes*, ou para descobrir se um arquivo pertence a um determinado conjunto de arquivos. Um arquivo em um sistema que nunca teve seu conteúdo alterado, sempre deverá ter um único número de *hash*. Na análise de *malwares*, todo arquivo analisado recebe uma identificação baseada no cálculo *hash*, seja ele um *malware* ou não, permitindo, além de identificá-lo, verificar a sua integridade. Uma vez atribuído o *hash* para o arquivo, o *malware* poderá ser rapidamente identificado. São exemplos de algoritmos *hash* utilizados para gerar a identificação digital, o MD5, o SHA-1, o SHA-256 e o SHA-512. Além do cálculo de *hash*, cálculo de *fuzzy hashes* podem também ser utilizados para determinar a similaridade entre arquivos.

A figura 2.2 é um extrato da análise que realizamos em nosso trabalho sobre um código malicioso. Podemos observar que vários antivírus identificam o arquivo como um *malware*. Cada um dos antivírus atribuem nomes diferentes para este malware mas o arquivo possui assinatura digital única. Estas assinaturas podem ter sido geradas pelos *hashes* MD5, SHA1, SHA-256, SHA-512 ou *fuzzy hash (ssdeep)*.

Uma *string* é uma sequência de caracteres. Um programa pode ter *strings* como uma URL, se um programa acessa um *site* da Internet para fazer um *upload* ou *download* ou o nome de uma chamada de sistema para apagar um arquivo, etc, e podem servir como uma evidência das intenções do emprego do *malware*. Podemos obter *strings* em um código usando um programa como o IDA Pro ou o pyew, ou de forma mais simples, podemos utilizar o aplicativo *strings*. Nem sempre será possível obter informações a partir de *strings* porque o código pode ter sido ofuscado. Em um programa ofuscado o autor do código procura tornar difícil ou mesmo evitar que se entenda o fluxo de execução do

TAB. 2.2: Serviços de Identificação de Malwares Baseados na Web

Web Service	Características
VírusTotal	verificação do arquivo por mais de 50 programas antivírus
	registra a primeira e a última data de verificação de um arquivo
	tamanho do arquivo
	hashes baseados em MD5, SHA1, SHA256
	Identificação do tipo de arquivo
	análise da estrutura de arquivos PE
	relatórios de segurança e referência cruzada de informações
	verificação de URLs
	permite realizar busca na base de dados do VírusTotal
	fórum de discussão para os usuários do VírusTotal
	permite submeter arquivos com o script vtsubmit.py (OBERHEID, 2014)
	http://www.vírustotal.com
VirScan	verificação do arquivo por 39 programas antivírus
	tamanho do arquivo
	hashes baseados em MD5, SHA1 para cada arquivo submetido
	http://www.virscan.org
Jotti	permite verificação do arquivo por 22 programas antivírus
	tamanho do arquivo
	hashes baseados em MD5, SHA1 para cada arquivo submetido
	tipo do arquivo e seu magic file
	identificação de empacotamento
	http://vírusscan.jotti.org/en
Metascan	permite verificação do arquivo por 40 programas antivírus
	registra a primeira e a última data de verificação de um arquivo
	tamanho do arquivo
	hashes baseados em MD5, SHA1, SHA256 para cada arquivo submetido
	identificação do tipo de arquivo
	identificação de empacotamento
	http://www.metascan-online.com

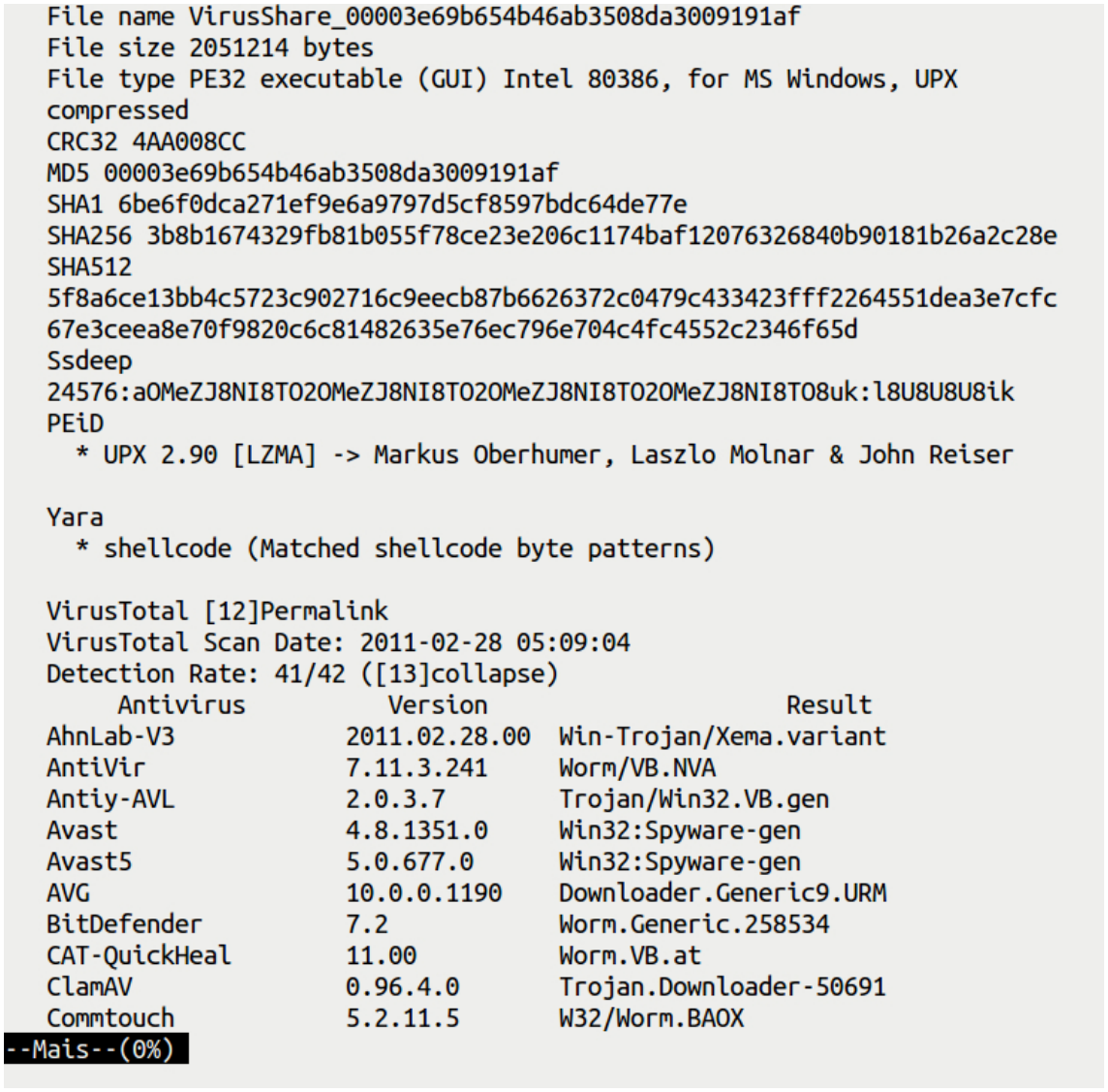


FIG. 2.2: Identificação digital de um arquivo

programa, o acesso à variáveis, dos valores guardados nos registradores e a visualização de *strings*.

A análise das informações obtidas na estrutura dos arquivos como, por exemplo, na estrutura de um arquivo com formato Win PE, podem ser úteis na identificação dos propósitos de um código. Esta análise permite obter informações sobre como o arquivo foi compilado, se o arquivo foi comprimido, criptografado ou ofuscado, listar as funções que se vinculam dinamicamente ao código e qual a funcionalidade das DLLs, quais funções foram exportadas e importadas, data da criação do arquivo, entropia dos dados, endereços suspeitos de entrada de seções, nome do arquivo, tamanho do arquivo, tipo do arquivo,

hash do arquivo, IAT suspeitos, etc.

A seguir são dadas algumas explicações destas características e suas importâncias para a análise de um código.

O cabeçalho de um arquivo PE possui informações sobre as funções que são utilizadas por um executável. Bibliotecas importadas para DLLs no sistema Windows são representadas pelos arquivos com extensão .lib. Exemplo: a DLL Kernel32.dll é uma biblioteca dinâmica utilizada para a execução de funções básicas no Windows, como, funções para criar um arquivo e gerenciar a memória. Esta biblioteca é vinculada por meio do arquivo kernel32.lib. A vinculação para bibliotecas dinâmicas é normalmente controlada pela vinculação a uma biblioteca de importação durante a fase de link-edição do programa. O executável criado conterá uma tabela de endereços importados ou *Import Address Table (IAT)*. Toda DLL referenciada contém sua própria entrada na tabela IAT. Desta forma, pode-se analisar por meio da IAT a existência de referências para funções que executem ações suspeitas. O programa pescanner.py pode ser utilizado para identificar estes casos (LIGH, 2010). Ele possui uma lista de 15 APIs que são consideradas suspeitas, mas é possível alterar o código e inserir outras. DLLs e programas executáveis .EXE podem exportar funções. O arquivo PE possui informações sobre quais DLLs um arquivo exporta. Uma DLL implementa uma ou mais funções e as exporta para o uso de um executável. EXEs não foram projetados para fornecer funcionalidades para outros EXEs de tal forma que funções exportadas por eles é algo raro e, portanto, suspeito. Empacotadores (*packers*) podem ser identificados utilizando o aplicativo PEiD para Windows ou o aplicativo pescanner.py. Um empacotador é um programa utilizado para ofuscar o código dificultando a sua desmontagem e análise (DEANDRADE, 2013). O programa PEiD contém assinaturas de empacotadores que podem ter sido utilizados. Permite que novas assinaturas sejam incluídas, se necessário, em sua base de dados. Outro recurso que pode ser utilizado é o aplicativo Yara. O Yara permite também criar arquivos de regras para a identificação de *packers* utilizados para ofuscar o código. Sua vantagem é que ele proporciona um método de plataforma cruzada (Windows, Linux, Mac OS) para a detecção de empacotadores. O tipo de um arquivo analisado pode ser identificado utilizando aplicativos como o file no Linux ou o pacote python-magic ou uma assinatura do Yara.

O arquivo de um *malware* pode possuir qualquer tamanho, desde alguns Kbytes até muitos Mbytes. Para permitir a sua fácil propagação na Internet, por meio de *e-mails*,

downloads e mensagens instantâneas, normalmente seus criadores desenvolvem código com poucos Kbytes. E, por isto, 97% dos *malwares* descobertos nos últimos cinco anos possuem menos de 1 MB (FORTINET, 2014). Entretanto, deve-se observar que alguns *malwares* mais complexos e com maior tamanho de arquivo têm surgido. Um exemplo é o Flame (w32.Flame.skywiper) (GOYAL, 2012), um *malware* desenvolvido com o objetivo de realizar espionagem e considerado o código malicioso mais sofisticado já desenvolvido. Após obter acesso, o atacante instala vários *backdoors* que lhe permitirão obter credenciais válidas e mover-se pela rede. Como o tamanho do arquivo é grande, o Flame possui forma de contágio semelhante à do Stuxnet, ou seja, por meio da possível utilização de um *pen-drive*.

O *time-stamp* de um arquivo identifica a data que o código foi compilado. Uma data estranha como, por exemplo, um arquivo com uma data de compilação inferior ao ano 2000 ou superior à data atual é considerado como suspeito. A entropia dos dados é uma heurística utilizada para detectar empacotamento. Valores altos de entropia indicam uma distribuição aleatória de *bytes* que compõem o executável, uma propriedade muito comum em dados comprimidos e cifrados. Um arquivo com alta entropia é considerado suspeito e cerca de 80% dos *malwares* possuem alta entropia (UGARTE-PEDRERO, 2012).

Na análise estática avançada uma inspeção mais profunda do código é realizada. Nesta análise o código é desmontado e é realizada uma engenharia reversa com o propósito de entender o fluxo das instruções do programa e as características do código. Um exemplo de programa que pode ser utilizado para desmontar um código de um *malware* é o IDA PRO (EAGLE, 2008). O Ida Pro é suportado por várias plataformas e pode reconhecer automaticamente o tipo do arquivo analisado, permite fazer a análise de funções de um programa, analisar o conteúdo de pilhas de execução, identificar variáveis locais, etc. Existem outros *softwares* como o pyew que é livre e que pode ser utilizado no ambiente Linux.

2.2.2.2 ANÁLISE DINÂMICA DE MALWARES

A análise dinâmica, assim como a análise estática, subdivide-se em análise dinâmica básica e análise dinâmica avançada. Na análise dinâmica básica o *malware* é executado em um ambiente virtualizado e o seu comportamento é monitorado, permitindo obter de forma automatizada as características de um código malicioso em execução. Também, a análise pode ser feita observando-se os estados do sistema operacional antes e após a

execução do código. Toda a análise baseia-se na observação das APIs que são executadas (WILLEMS, 2007). (SIKORSKI, 2012) define o seguinte conjunto de APIs que podem ser observadas: 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, ReadProcessMemory, 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.

A análise dinâmica requer um ambiente seguro e isolado. Para construir este ambiente utiliza-se uma técnica conhecida como *sandbox* que permite executar o código em um ambiente confinado. São exemplos de implementações de *sandbox* (OKTAVIANTO, 2013): virtualização do sistema operacional ou *jail* (exemplo: *virtual hosting*); execução

baseada em regras (exemplo: *frameworks* de segurança Apparmor e SELinux para o ambiente Linux); emulação de máquinas virtuais (exemplo: QEMU e Bochs); sandboxing (exemplo: Cuckoo Sandbox).

Para a análise de *malwares* o ambiente que utiliza *sandboxing* é muito empregado e permite observar as APIs acima citadas. Podemos dividir os *sandboxes* para análise de *malwares* em dois tipos: *sandbox on-line* e *sandbox standalone*.

Um *sandbox on-line* é um ambiente de *sandbox* para análise de *malwares* montado por uma organização que disponibiliza o serviço na Internet. O usuário do serviço não pode interferir nas configurações que lhe foram definidas pelo provedor. O provedor fornece normalmente uma *interface* para o usuário submeter o arquivo para análise. Ao final da análise é gerado o resultado com as informações e a situação do arquivo.

São exemplos de *sandboxes on-line*:

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

Um *sandbox standalone* é totalmente configurado e manutenido por quem o implementou. Para implementá-lo é necessário configurar máquinas virtuais que criem o ambiente isolado e seguro. Também é necessário utilizar um sistema como o Cuckoo Sandbox (OKTAVIANTO, 2013) para gerenciar as atividades de análise e geração de relatórios.

São exemplos de *sandboxes standalone*:

- Cuckoo Sandbox (http://www.cuckoosandbox.org/);
- Sandboxie (http://www.sandboxie.com/);
- Buster Sandbox (http://bsa.isoftware.nl/);

- Remnux (http://zeltser.com/remnux/);
- Zero Wine (http://zerowine-tryout.sourceforge.net/).

O Cuckoo Sandbox é um sistema para análise dinâmica de *malwares*, seu código é aberto e permite fazer a análise de *malwares* que executam em sistemas Windows (OKTAVIANTO, 2013). Permite que os seguintes tipos de arquivos sejam analisados: executáveis do Windows, DLLs, PDFs, documentos gerados pelo pacote Office, URLs, scripts PHP, arquivos de imagens (GIF, JPEG, etc).

Permite capturar informações de traços de chamadas e APIs executadas por todos os processos iniciados pelo *malware*, arquivos que foram criados, apagados e baixados pelo *malware* durante a execução, *dumps* de memória dos processos do *malware*, tráfego de rede em formato PCAP, fotografias das telas do sistema tiradas durante a execução do *malware* e *dumps* de memória completos de todas as máquinas virtuais.

A análise dinâmica avançada emprega a depuração (*debugging*) para verificação do código (SIKORSKI, 2012) (LIGH, 2010). Nesta técnica, utiliza-se um programa depurador (*debugger*) para testar e depurar um código. Existem dois tipos de depuração: a depuração do código fonte e a depuração do código assembly. A depuração do código assembly é a utilizada na análise de *malwares* porque não requer o acesso ao código fonte. Existem duas formas de depurar um programa. A primeira consiste em iniciar o programa com um programa depurador. Quando o programa é iniciado e carregado na memória, ele é imediatamente interrompido logo no ponto de entrada da execução. A partir deste ponto, obtém-se o controle completo dos passos de execução do programa depurado. Outra forma de depurar o programa é iniciar a depuração a partir de outro endereço de um programa em execução e não no inicial, de entrada. Neste caso, todas as *threads* do programa são pausadas e ele pode ser depurado. Esta é uma boa abordagem para depurar um processo que é afetado por um *malware*. Quando se utiliza um depurador, para examinar cada instrução do código e manter o controle da execução do depurador, utilizam-se as características de *single-stepping* e *step-into* que permitem examinar cada uma das instruções e funções chamadas pelo código. Outros recursos como pontos de quebra (*breakpoints*) e exceções (*exceptions*) são utilizados. *Breakpoints* são utilizados para interromper a execução do programa. Isto é necessário porque não é possível acessar os valores dos registradores ou os endereços de memória enquanto o programa está em execução.

As exceções também permitem manter o controle da execução do depurador. São úteis para tratar de questões como acessos inválidos à memória e divisões por zero. Podem também ser utilizadas para governar o fluxo de execução de um programa sem o envolvimento do depurador.

São exemplos de *softwares* depuradores: W32DASM, IDA, WinDbg, SoftICE e Ollydbg.

As técnicas de análise estática e dinâmica possuem pontos fracos que permitem aos desenvolvedores de *malwares* empregar técnicas de antianálise que impedem a observação das características de um *malware*.

2.2.3 TÉCNICAS DE ANTIANÁLISE DE MALWARES

Os desenvolvedores de *malwares* utilizam algumas técnicas de evasão com o objetivo de dificultar ou impedir a desmontagem do código necessária para se realizar a análise estática, dificultar ou impedir a execução da análise dinâmica automática em ambientes virtualizados pela detecção de máquinas virtuais e dificultar ou impedir a depuração de código para se realizar a análise dinâmica avançada. Mecanismos antianálise podem ser divididos em 4 categorias (BRANCO, 2012):

- Ofuscação: são técnicas utilizadas para tornar mais difícil a criação de assinaturas e a análise do código desmontado.
- Antidisassembly: são as técnicas utilizadas para comprometer o funcionamento dos desmontadores e/ou prejudicar o processo de desmontagem de código.
- Anti-VM: são técnicas utilizadas para detectar a presença de máquinas virtuais ou prejudicar o seu funcionamento.
- Antidebugging: são as técnicas utilizadas para comprometer o funcionamento de depuradores e/ou o processo de depuração do código dificultando a engenharia reversa.

2.2.3.1 OFUSCAÇÃO

A principal técnica de detecção de vírus é baseada na busca por assinaturas. Ela consiste na procura de uma sequência de *bytes* (assinatura) conhecida do vírus em uma base de dados (base de assinaturas). Na tentativa de dificultar a identificação de um *malware*, os

escritores de códigos maliciosos têm empregado diferentes técnicas de ofuscação (COZZOLINO, 2012). Uma técnica de ofuscação tem como objetivo dificultar ou impedir a análise estática. Com este objetivo os criadores de *malwares* podem empregar uma variedade de técnicas de ofuscação. Estas técnicas de ofuscação evoluem acompanhando a evolução dos *malwares* e (SZOR, 2005), de acordo com estas técnicas, classifica os *malwares* da seguinte forma:

- Malwares Cifrados;
- Malwares Oligomórficos;
- Malwares Polimórficos;
- Malwares Metamórficos.

Malwares cifrados empregam técnicas de cifragem de código para evitar a detecção baseada em assinaturas pelos antivírus. Nesta abordagem, um *malware* cifrado é composto de um decodificador e de um corpo principal cifrado (YOU, 2010).

Para que o arquivo infectado seja executado, ele deve ser decifrado por um decifrador. Para cada infecção, o *malware* cria uma parte cifrada única, ocultando assim a sua assinatura. O principal problema desta abordagem é que o decifrador permanece constante de geração a geração, tornando possível aos mecanismos antivírus detectar os *malwares* que utilizam um determinado decifrador.

A fim de fazer frente a esta deficiência, os criadores de *malwares* desenvolveram tecnologias de mutação (YOU, 2010), que provocaram mudanças no código de geração a geração, e que foram aplicadas aos códigos dos decifradores. Desta forma, de acordo com a evolução dos *malwares*, os criadores desenvolveram os **malwares oligomórficos** que possuíam uma capacidade pequena de alterar seus decifradores, ou seja, geravam poucas formas (*oligo = pouco, morfo = forma*). Como os malwares oligomórficos podiam gerar poucos decifradores, eles ainda podiam ser facilmente detectados por assinaturas.

Para enfrentar esta limitação, os desenvolvedores de *malwares* criaram os **malwares polimórficos** *(poli = muitos, morfo = forma).*

Os malwares polimórficos podiam gerar muitos decifradores distintos com a ajuda de técnicas de codificação como a inserção de código inerte, a troca de registradores, e outras técnicas. Um *malware* polimórfico não deve conter partes que permaneçam idênticas entre infecções, tornando-o muito difícil de ser detectado usando assinaturas. Também ocorreu

o surgimento de poderosos *softwares* para geração de código com mutação, os geradores polimórficos Dark Avenger's Mutantion Engine (CHEN, 2004). Os geradores polimórficos auxiliavam os criadores de *malwares* a coverter facilmente seus códigos para uma versão polimórfica. Embora os *malwares* polimórficos pudessem efetivamente impedir a verificação por assinaturas, seus corpos principais constantes apareciam após a decifração do código, sendo uma importante fonte para a detecção. Ferramentas antivírus foram criadas e que exploravam esta vulnerabilidade, empregando técnicas de emulação (YOU, 2010). Nesta técnica o *malware* era executado em um emulador, chamado de *sandbox*. Uma vez que o corpo decifrado do *malware* fosse carregado na memória, utilizava-se a detecção convencional baseada no método de assinaturas para detectá-lo. Para detectar e evitar esta emulação, os *malwares* passaram a utilizar técnicas de blindagem (*armored vírus*) (SZOR, 2005) (GASPAR, 2007). Entretanto, a evolução dos produtos antivírus tornou os *malwares* polimórficos obsoletos e totalmente detectáveis. A próxima evolução dos *malwares* foram os **malwares metamórficos** *(meta = mudança, transformação; morfo = forma)*. Os *malwares* metamórficos aperfeiçoaram o emprego das técnicas de ofuscação, aplicando-as aos seus corpos, fazendo com que os *malwares* das diferentes gerações pareçam ser diferentes, mas que funcionam identicamente (KONSTANTINOU, 2008). Para que ocorra uma evolução, o *malware* metamórfico deve ser capaz de reconhecer, analisar e produzir a mutação de seu próprio corpo quando ele se propaga. A detecção de um *malware* metamórfico que não revela o seu corpo em memória é muito difícil para os mecanismos de detecção dos antivírus.

Seguem alguns exemplos de técnicas de codificação utilizadas na criação dos *malwares* metamórficos para criar novas gerações com aparências diferentes, mas com funcionalidades idênticas (KONSTANTINOU, 2008):

- inserção de código inerte: é uma técnica simples utilizada por muitos *malwares* polimórficos e metamórficos para produzir a evolução do código. Nesta técnica o criador do *malware* insere código que não produz impacto algum na funcionalidade do *malware*.

- troca de registradores: este método foi utilizado pelo vírus Win95/Regswap criado por Vecna 1998. Diferentes gerações do vírus usariam o mesmo código, mas empregavam registradores diferentes na execução (FERRIE, 2001).

- técnicas de permutação: nesta técnica o código é constante e a metamorfose é al-

cançada dividindo o código em blocos aleatórios e conectando-os com instruções de desvio que mantêm o controle e fluxo do processo. Um exemplo é o *malware* Win32/Ghost que foi descoberto em 2000 e tinha a capacidade de reordenar suas rotinas de geração a geração, sendo a reordenação das gerações de ordem fatorial. Possuía 10 sub-rotinas e, como o seu número de gerações era igual a $n!$, então, o número total de gerações diferentes para este *malware* era igual a 3628800 (KONSTANTINOU, 2008).

- Inserção de instruções de salto: nesta técnica o *malware* insere e remove instruções de salto (*jump*) dentro de seu código e a cada instrução de salto apontará para uma nova instrução do *malware*. O *malware* nunca gera um código constante em seu corpo, nem mesmo em memória, e assim, a detecção do *malware* utilizando a busca por *strings* é virtualmente impossível (FERRIE, 2001).

- substituição de instruções: nesta técnica o *malware* substitui algumas instruções suas por outras equivalentes. O vírus Win95/Zmist, por exemplo, utilizava as seguintes substituições em seu código (FERRIE, 2001): reversão de condições de salto, *moves* para registradores substituídos por sequências *push/pop*, codificação alternativa de *opcode* e intercambiamento entre *xor/sub* e *or/test*.

- mutação de códigos do *host*: o *malware* possui a capacidade de mudar a si mesmo, produzindo novas gerações, e a outros códigos de seu sistema hospedeiro. Desta forma, o vírus consegue gerar novos vírus e *worms*. Para fazer isto, o *malware* utiliza randomicamente uma rotina de formação de código. Também, porque o endereço de entrada da aplicação pode ser diferente, a desinfecção não pode ser perfeita. O Win95/Bistro é um exemplo de *malware* que empregava esta técnica.

- integração de código: o vírus Win95/Zmist utilizava esta técnica. O vírus dividia um arquivo PE em seus menores elementos e necessitava de 32 MB de memória para que pudesse ser executado. O vírus quebrava o código atacado em partes, injetava-se nele, em qualquer espaço disponível, gerava novamente o código e suas referências de dados e gerava finalmente o executável (FERRIE, 2001).

2.2.3.2 ANTIDISASSEMBLY

Esta técnica utiliza código ou dados criados com o objetivo de que ferramentas *disassembly* produzam listagens incorretas (SIKORSKI, 2012). Os criadores dos *malwares* utilizam *antidisassembly* para evitar a engenharia reversa do código e, assim, retardar ou evitar a análise do código malicioso. As técnicas *antidisassembly* podem ser bastante genéricas e aplicadas a vários programas desmontadores e podem ser bastante específicas para desmontadores específicos.

As técnicas *antidisassembly* podem ser empregadas devido aos pontos fracos dos algoritmos de desmontagem.

Existem dois tipos de algoritmos de desmontagem (LINN, 2003): os algoritmos lineares e os orientados a fluxos.

Os lineares são os mais fáceis de implementar e de apresentar problemas. Os algoritmos lineares ou de varredura linear iniciam a desmontagem do código em um determinado *byte* (por exemplo, o primeiro *byte* definido como o início do programa) e segue a partir deste ponto *byte-a-byte* até um término predefinido (o endereço de término de uma seção PE). A principal desvantagem desta abordagem é que os dados colocados no meio das instruções de código podem gerar algum ruído, porque eles podem ser interpretados como código (BRANCO, 2012). Um exemplo de programa *desmontador* que utiliza este método é o Objdump (STEVANOVIC, 2014).

Os algoritmos orientados a fluxos ou recursivos transversais seguem o fluxo do programa ao invés de seguir a desmontagem *byte-a-byte*. Sua desvantagem é que nem sempre será possível predizer de forma estática (ou seja, sem executar o código) o fluxo do programa de forma exata (BRANCO, 2012). Isto pode ter como resultado que algumas partes do código não poderão ser desmontadas e também algum ruído poderá ser gerado. Algumas áreas do código que não forem alcançadas poderão ser submetidas a um processamento de fluxo linear e esta variação é chamada de desmontagem especulativa (*speculative disassembly*). Um exemplo de *desmontador* que utiliza a abordagem recursiva transversal é o IDA PRO. São exemplos de técnicas *antidisassembly* (BRANCO, 2012): *garbage bytes*, alteração do controle de fluxo do programa, saltos condicionais falsos, redirecionamento do fluxo para o meio de uma instrução.

- *garbage bytes*: esta técnica consiste na adição de *bytes* adicionais que nunca serão executados e que servem para quebrar o alinhamento do algoritmo linear, prejudi-

cando o seu entendimento. Os algoritmos transversais recursivos também podem ser comprometidos pela introdução de *garbage bytes* se ocorrer uma situação que possa ser forçada devido à existência de um mesmo conjunto de *bytes* com mais de uma interpretação.

- alteração do controle de fluxo do programa: esta técnica baseia-se na alteração forçada incondicional do fluxo de programa. Uma instrução *assembly* JMP pode ser usada para implementar esta técnica. Desta forma, parte do código pode ficar sem ser analisado e comprometido.

- saltos condicionais falsos: esta técnica baseia-se na criação de saltos condicionais cujas condições sempre sejam as mesmas.

- truques de chamadas ou *call tricks*: esta técnica baseia-se na alteração do endereço de retorno padrão de uma instrução. Quando utilizada em conjunção com outras técnicas como, por exemplo, *garbage bytes*, pode prejudicar todos os tipos de desmontadores.

- redirecionamento para o meio de uma instrução: esta técnica baseia-se no redirecionamento do fluxo de programa para o meio de uma instrução. Pode comprometer tanto o algoritmo linear quanto o orientado a fluxo. Um exemplo seria a ocultação de uma instrução no meio de outra. Desta forma, o desmontador mostraria uma instrução que não é executada no lugar da instrução que reside no meio de seus *bytes*.

2.2.3.3 ANTI-VM

Para detectar *malwares* blindados, *zero-day malwares*, *malwares* metamórficos, pode-se utilizar técnicas de virtualização e *sandbox*. A análise dinâmica de *malwares* utiliza técnicas de virtualização em um processo automatizado de coleta de informações de um programa em execução, permitindo que seja feita a sua análise e determinando se ele é ou não um *malware*. Os criadores de *malwares* tentam prejudicar ou impedir a identificação por sistemas que empregam estas técnicas utilizando métodos anti-vm. Desta forma, *malwares* que empregam técnicas anti-vm, não executarão em ambientes virtualizados (OKTAVIANTO, 2013).

Para demonstrar um exemplo de técnica utilizada por criadores de *malwares* para identificar um ambiente virtualizado, utilizaremos o programa pafish, que pode ser baixado juntamente com o código fonte no endereço: https://github.com/a0rtega/pafish. A ferramenta implementa técnicas frequentemente utilizadas em *malwares* para detectar máquinas virtuais.

O programa pafish utiliza alguns sensores que permitem detectar um ambiente virtualizado, a utilização de emuladores e de *sandboxes*.

O programa que instalamos no Windows XP, na máquina virtual que utilizamos nos experimentos, foi capaz de detectar a presença de depuradores, de *sandboxes*, de emuladores e de *hook functions*. A figura 2.3 mostra a saída do programa pafish. Podemos observar: que o programa detectou a presença de depuradores, executando as funções IsDebbuger-Present() e OutputDebugString(); o *sandbox* instalado, utilizando a dll sbiedll.dll; *hooking functions*, que podem ser utilizadas para estender uma funcionalidade do código; o ambiente de virtualização do VirtualBox pela inspeção de registros do sistema.

Na saída do programa pafish, pode-se ver que foi possível detectar algumas características que identificam a presença de emuladores, máquinas virtuais e *sandbox* no sistema, utilizando a API GetProcAddress da biblioteca Kernel32.dll.

Para detectar uma máquina virtual, por exemplo, o programa utiliza o módulo vbox.c. O trecho de código escrito em linguagem C é apresentado na figura 2.4. Nele, podemos observar a utilização da função RegOpenKeyEx para abrir a chave de registro. Se for encontrado o identificador, a função retornará ERROR_SUCCESS, caso contrário um valor diferente de zero será retornado. Se ERROR_SUCCESS foi retornado, a chave de registro "Identifier" é lida e armazenada no vetor *value*. Todos os caracteres da *string* são convertidos para caracteres maiúsculos e finalmente a sequência armazenada em *value* é comparada com a *string* "VBOX". Se forem iguais, a função retorna 0, indicando que o sistema está executando sobre o ambiente VirtualBox.

2.2.3.4 ANTIDEBUGGING

A técnica de *antidebugging* é uma técnica antianálise popular e utilizada por *malwares* para reconhecer quando existe um depurador de código que controla a execução (SIKORSKI, 2012). Uma vez que o *malware* detecta a presença de um depurador, ele pode alterar o seu curso normal ou interromper imediatamente a sua execução, dificultando o entendimento do funcionamento do *malware* e aumentando o tempo necessário para a sua

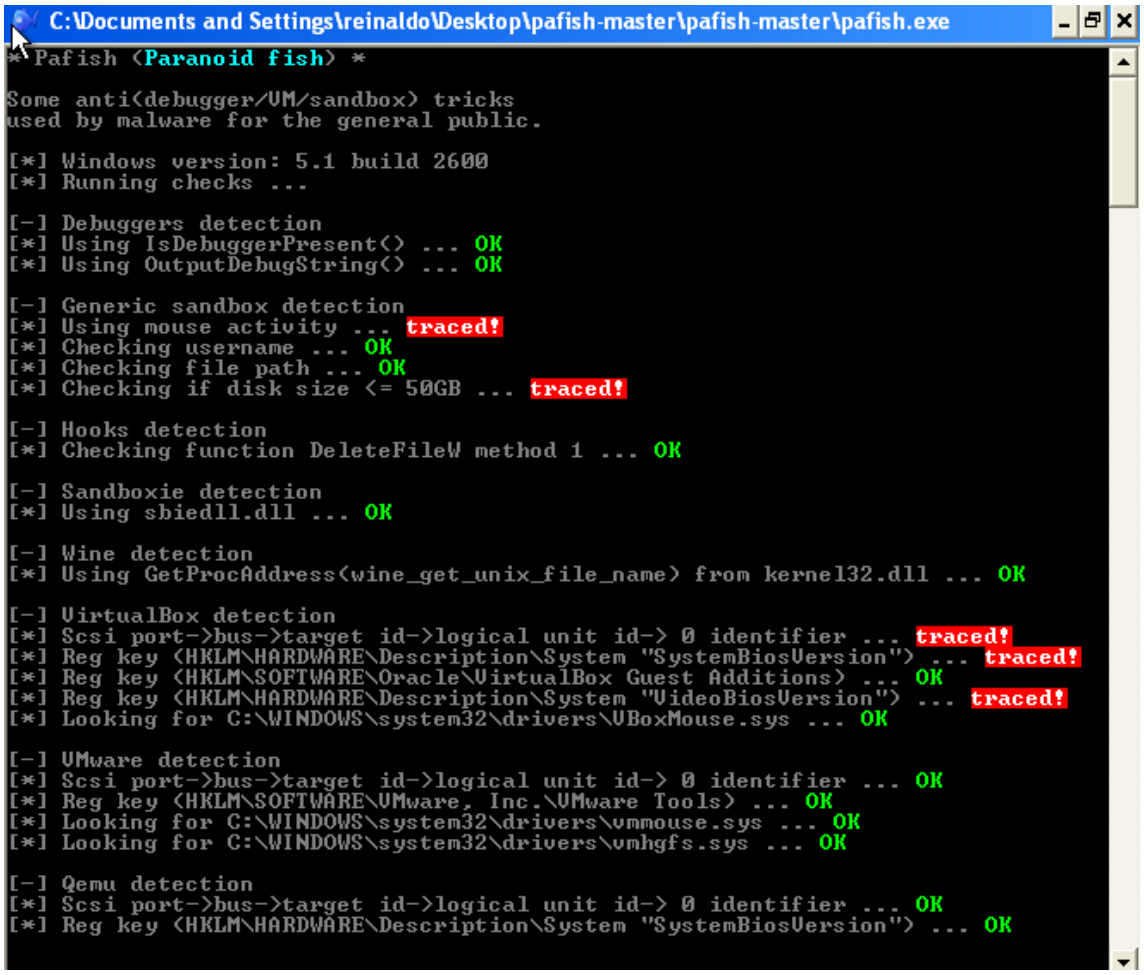


FIG. 2.3: Programa pafish.exe identificando uma máquina virtual

identificação. Existem várias, talvez centenas de técnicas *antidebugging*. Estas técnicas podem utilizar métodos para a detecção de depuradores no ambiente Windows, métodos que identifiquem o comportamento do depurador, métodos que interfiram na funcionalidade do depurador e métodos que explorem as vulnerabilidades do depurador (BRANCO, 2012).

No método de detecção de depuradores no ambiente Windows pode se verificar o emprego de APIs no código do *malware* como, por exemplo, as APIs IsDebuggerPresent, CheckRemoteDebuggerPresent, NtQueryInformationProcess, OutputDebugString, verificar os valores de estruturas de dados como a flag BeingDebbuged, a flag ProcessHeap, a flag NTGlobalFlag e pode-se verificar resíduos de depuradores no sistema disponíveis no registro do Windows como, por exemplo, a presença do valor de registro HKEY _ LOCAL _ MACHINE / SOFTWARE / Microsoft / Windows / CurrentVersion/AeDebug.

```
int vbox_reg_key1() {
    HKEY regkey;
    LONG retu;
    char value[1024];
    int i;
    DWORD size;
    size = sizeof(value);
    retu = RegOpenKeyEx(HKEY_LOCAL_MACHINE, "HARDWARE\\DEVICEMAP\\
       Scsi\\Scsi Port 0\\Scsi Bus 0\\Target Id 0\\Logical Unit Id
       0", 0, KEY_READ, &regkey);
    if (retu == ERROR_SUCCESS) {
        retu = RegQueryValueEx(regkey, "Identifier", NULL, NULL, (
           BYTE*)value, &size);
        if (retu == ERROR_SUCCESS) {
            for (i = 0; i < strlen(value); i++) { /* case-
               insensitive */
                value[i] = toupper(value[i]); }
            if (strstr(value, "VBOX") != NULL) {
                return 0; }
            else {
                return 1; }
        }
        else {
            return 1; }
    }
    else {
        return 1;
    }
}
```

FIG. 2.4: Programa vbox.c

No método de identificação do comportamento do depurador o *malware* pode utilizar as técnicas INT *scanning, checksum checks e timing checks*.

A técnica INT *scanning* consiste em identificar a presença da interrupção de *software* INT 3 utilizada pelos depuradores para temporariamente substituir uma instrução em um programa em execução e chamar um manipulador de exceção do depurador, sendo este o mecanismo utilizado quando são definidos *breakpoints* pelo analista para depurar o código.

A verificação de *checksum* pode ser empregada como um meio alternativo à busca de interrupções. Neste caso, o *malware* pode calcular um *checksum* sobre uma seção de seu código para descobrir a presença do depurador.

Timing Checks ou verificações de tempo é a mais popular forma utilizada para detectar um depurador e explora a lentidão de execução dos processos executados sobre o controle do depurador.

No método de interferência sobre a funcionalidade do depurador o *malware* pode utilizar as seguintes técnicas para interferir nas operações normais do depurador: *thread local storage (TLS) callbacks*, exceções e inserção de interrupções.

Muitos depuradores iniciam um programa no endereço de ponto de entrada deste programa conforme indicado no PE Header. Uma TLS pode ser utilizada para executar o código do *malware* antes deste ponto de entrada. Quando uma TLS é implementada, o código conterá uma seção .tls no PE Header, indicando que o código possui uma característica suspeita de *antidebugging*.

Interrupções geram exceções que são utilizadas pelo depurador para executar *breakpoints*. Exceções podem ser usadas para atrapalhar ou detectar um depurador. As principais detecções baseadas em exceções baseiam-se no fato de que o depurador irá interceptar a exceção e não a passará imediatamente para o processo que está sendo depurado para manipulação. Se o depurador não passa a exceção para o processo, isto pode ser detectado.

Uma forma clássica de *antidebugging* é o uso de exceções por meio da inserção de interrupções no meio de uma sequência de instruções válidas. Dependendo da configuração do depurador estas interrupções poderão interromper o funcionamento do depurador.

No método de exploração de vulnerabilidades, o *malware* explora vulnerabilidades existentes no *software* do depurador com o objetivo de impedir a depuração.

3 APRENDIZADO DE MÁQUINA

As análises estática e dinâmica permitem que o analista obtenha uma série de características dos códigos, possibilitando-lhe fazer a sua análise, para determinar se um arquivo deve ser classificado como *malware* ou não *malware*. Trata-se de um processo humano que pode ser automatizado utilizando uma técnica de inteligência artificial chamada de aprendizado de máquina. A combinação das análises estática e dinâmica e a utilização de aprendizado de máquina podem melhorar a capacidade e velocidade de detecção dos *malwares* e é uma possível boa solução para tratar com *malwares* blindados, metamórficos, ou aqueles ainda não identificados *(zero-day malwares)*.

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

A indução é um processo de generalização pela observação e combinação de exemplos particulares. Na indução, um conceito é aprendido efetuando-se inferência indutiva sobre os exemplos apresentados. Portanto, as hipóteses geradas por meio da inferência indutiva podem ou não apresentar a verdade (MONARD, 2003).

Cuidados devem ser tomados quando se utiliza métodos indutivos, pois se o número de exemplos for insuficiente, ou se os exemplos não forem bem escolhidos, as hipóteses obtidas podem ser de pouco valor. Assim, a qualidade do aprendizado depende da qualidade dos dados inicialmente fornecidos ao algoritmo de aprendizado e esta qualidade pode ser verificada por métricas estatísticas. Os aprendizados indutivos utilizados pelos métodos de aprendizado de máquina podem ser o aprendizado supervisionado e o aprendizado não supervisionado. Além dos aprendizados supervisionado e não supervisionado, também existe o aprendizado por reforço.

No aprendizado supervisionado o indutor, que é o algoritmo de aprendizado de máquina, a partir de um conjunto de exemplos, gera as hipóteses (classificações) dos exemplos de um conjunto. O indutor recebe um conjunto de exemplos de treinamento constituído por várias instâncias que possuem atributos e rótulos de classe e que ao final do processo de indução, recebem uma predição de classe.

No aprendizado não supervisionado, o indutor agrupará os exemplos em grupos ou clusters que posteriormente serão analisados para identificação do significado dos grupos.

O aprendizado por reforço é muito utilizado pela robótica e consiste na observação do ambiente para adquirir aprendizado.

Para a classificação dos *malwares*, o aprendizado supervisionado é muito utilizado. Assim, por exemplo, no caso da classificação de *malwares*, temos um conjunto de exemplos formado por um vetor de características, ou atributos, e cada instância deste conjunto, possui uma classificação como *malware* ou *não malware*. Ao final da indução cada uma das instâncias do conjunto de exemplos recebe uma predição de classe dada pelo indutor como maligno (*malware*) ou benigno (*não malware*).

A figura 3.1 ilustra o processo genérico do aprendizado de máquina supervisionado. Nele, o especialista, que possui o domínio do conhecimento, definirá quais são os atributos e os rótulos de classe, bem como seus valores iniciais. Os dados classificados pelo analista são submetidos ao indutor que fará a classificação. Após a classificação do indutor, os resultados obtidos são analisados e ajustes são feitos com o objetivo de melhorar a precisão do processo.

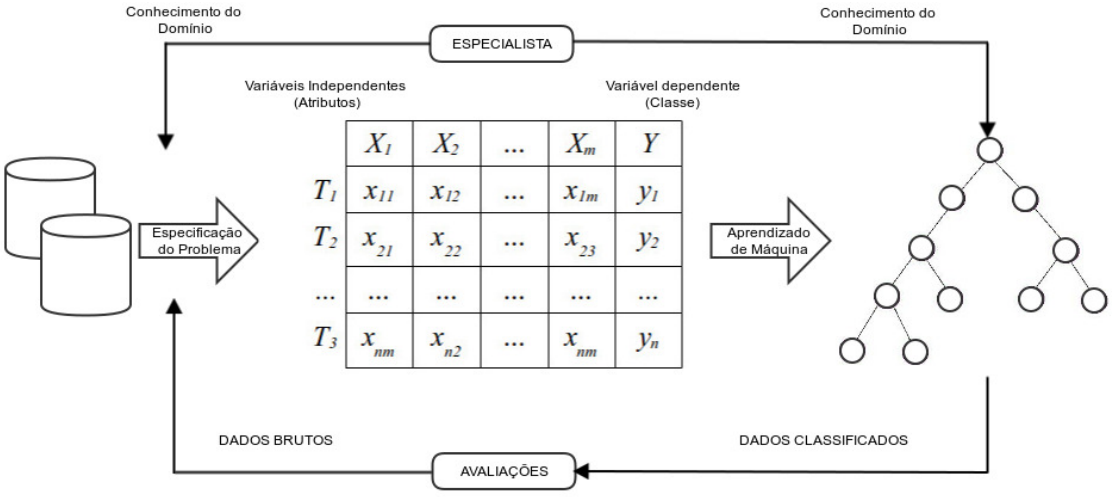


FIG. 3.1: Processo de Aprendizado de Máquina Supervisionado
Fonte: (MONARD, 2003)

O aprendizado de máquina supervisionado pode ser de dois tipos: classificação e regressão. Quando os valores da classe são nominais, discretizados, o aprendizado é uma classificação e quando são reais, contínuos, o aprendizado é uma regressão.

Os atributos podem ser nominais (discretos) e contínuos. Exemplo de atributo dis-

creto: data_suspeita: sim, não; Exemplo de atributo contínuo: Tamanho_arquivo. Quando um valor de um atributo for desconhecido, utiliza-se ?. Um ponto importante a ser considerado é a escolha de atributos com boa capacidade preditiva.

Uma **classe** é um rótulo que descreve o fenômeno de interesse. Exemplo: um arquivo pode ser um malware ou um não malware.

Um **exemplo**, também chamado de caso, registro ou dado é uma tupla de valores de **atributos (características)**.

Exemplos são tuplas $T_i = (x_{i1}, x_{i2}, ..., x_{im}, y_i) = (\vec{x}_i)$, também denotados por (x_i, y_i). x_i é um vetor de atributos e y é uma classe, tal que $y_i \in \{C_1, C_2, \cdots, C_n\}$.

Um **conjunto de exemplos** é constituído por vários exemplos que possuem atributos comuns. Uma instância possui atributos e uma classe associada. Em um conjunto de exemplos T com n exemplos e m atributos, a linha i refere-se ao *i-ésimo* exemplo ($i = 1, 2, 3, ...n$) e a entrada x_i refere-se ao valor do *j-ésimo* ($j = 1, 2, 3...., m$) atributo X_j do exemplo i.

Um indutor gera uma classificação ou hipótese a partir de um conjunto de exemplos de treinamento, procurando predizer com a maior precisão possível a classe de um novo exemplo.

Formalmente, em classificação, um exemplo é um par $(x_i, f(x_i))$ onde x_i é a entrada e $f(x_i)$ é a saída. O indutor induz uma função h que aproxima f, normalmente desconhecida. h é chamada **hipótese** sobre a função objetivo f, ou seja, $h(x_i) \equiv f(x_i)$. Alguns autores costumam também chamar a hipótese de modelo.

3.1 ALGORITMOS DE APRENDIZADO DE MÁQUINA

O aprendizado de máquina baseia-se em vários padrões estruturais. (MONARD, 2003) define que estes padrões podem ser assim resumidos:

- Sistemas de aprendizado simbólicos: utilizam estruturas de representação simbólicas de um conceito, em seu processo de aprendizado, realizando a análise de exemplos e contraexemplos do conceito. Estas estruturas normalmente estão representadas na forma de alguma expressão lógica, árvore de decisão, regras ou rede semântica.

- Sistemas de aprendizado estatístico: utilizam modelos estatísticos como, por exemplo, o aprendizado bayesiano, para encontrar uma boa aproximação de um conceito induzido.

- Sistemas de aprendizado baseado em exemplos: classificam exemplos nunca vistos por meio de exemplos similares conhecidos (Exemplo: Nearest Neighboors e Raciocínio Baseado em Casos).

- Sistemas de aprendizado conexionista: sua estrutura se inspira no modelo biológico do sistema nervoso. Exemplo: Redes Neurais.

- Sistemas de aprendizado evolucionista: deriva-se do modelo biológico de aprendizado, procurando reproduzir no sistema um comportamento inspirado na Teoria de Darwin de seleção natural. Exemplo: algoritmos genéticos.

O Naive Bayes, por exemplo, é um algoritmo de aprendizado de máquina que emprega o aprendizado estatístico para a classificação de exemplos.

O algoritmo classificador Naive Bayes, origina-se da Teoria de Bayes e é usado na modelagem preditiva. O algoritmo não leva em consideração as dependências que possam existir entre os dados e, por isto, é chamado de ingênuo (Naive). Isto ocorre porque os atributos são condicionalmente independentes, indicando que a informação de um evento nada diz em relação a outro evento.

(RUSSELL, 1995) também o denomina de classificador *bayesiano*. O Naive Bayes considera que, tendo exemplos de cada classe, pode-se, por meio de inferência, identificar o processo gerador da classe.

Outro exemplo de algoritmo de aprendizado de máquina é o SVM. Máquina de Vetores de Suporte ou Support Vector Machine (SVM) é um algoritmo de aprendizado supervisionado que permite analisar dados e reconhecer padrões e é utilizado para classificação e regressão. Introduzida por meio da teoria estatística de aprendizagem, a classificação é baseada no princípio da separação ótima entre classes (CORTES, 1995).

O modelo SVM pode ser formulado para definir fronteiras lineares e não lineares para a separação do conjunto de dados. A escolha do classificador, linear ou não linear, depende da natureza do problema. SVMs lineares obtém fronteiras lineares para a separação de dados pertencentes a duas classes. Por meio do algoritmo SVM busca-se encontrar um hiperplano ótimo que separe as classes de dados com a maior margem possível. A figura 3.2 representa a solução para um problema de classificação linearmente separável em dois conjuntos de hipóteses: Class 1 são as bolas brancas e Class 2 são as bolas pretas.

No mundo real, na maioria das vezes, não será possível realizar a classificação linear ainda que se utilize a margem de folga. Então, neste caso, os dados serão mapeados em

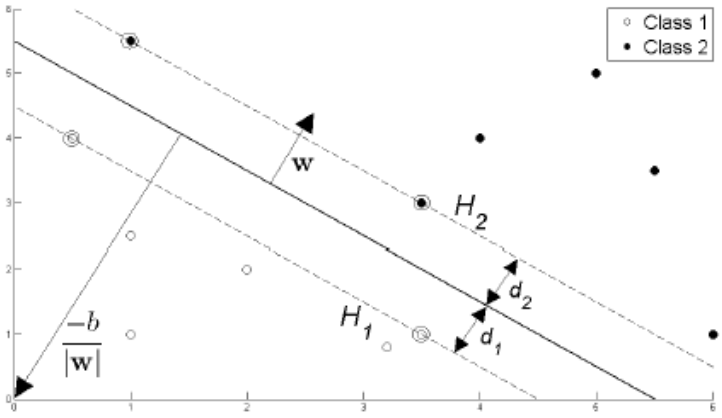


FIG. 3.2: SVM. Separação linear de hipóteses
Fonte: (LORENA, 2007)

um espaço de dimensão maior. A figura 3.3 mostra um exemplo de representação para dados não lineares. Observa-se que o problema pode ser resolvido aumentando a dimensão de espaço para o mapeamento dos dados.

A única informação necessária para realizar o mapeamento é a de como realizar o cálculo de produtos escalares entre dados nesse espaço de características. Isto é obtido com o uso de funções denominadas Núcleos ou *Kernels*. Um *kernel* K é uma função que recebe dois pontos x_i e x_j do espaço de entradas e computa o produto escalar desses dados (LORENA, 2007). Tem-se, de modo geral:

$$k(x_i, x_j) = \varphi(x_i) * \varphi(x_j)$$

Neste trabalho, destacamos o uso de sistemas de aprendizado simbólicos, mais precisamente, o uso de algoritmos baseados em árvores de decisão, como o ID3, o C4.5 e o C5.0 e o uso de um algoritmo de comitê constituído de várias árvores, o Random Forest.

3.2 ALGORITMOS BASEADOS EM ÁRVORES DE DECISÃO: ID3, C4.5 E C5.0

Existem algoritmos de aprendizado de máquina que utilizam como modelo de representação as árvores de decisão. Estes algoritmos de indução sobre árvores pertencem à família de algoritmos *Top Down Induction of Decision Trees - TDIDT* (MONARD, 2003). O modelo de árvore de decisão utiliza o método dividir para conquistar e podemos tomar

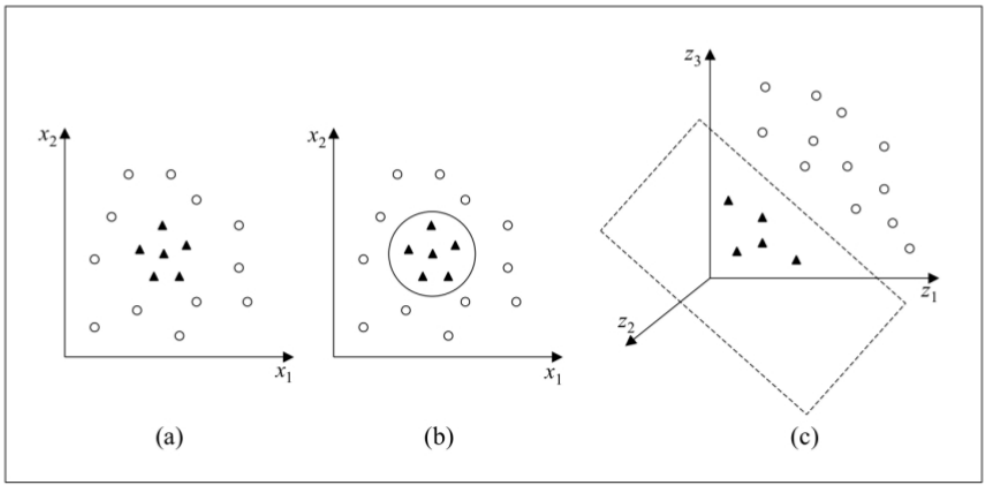


FIG. 3.3: SVM. Separação não linear de hipóteses. (a) Conjunto de dados não linear. (b) Fronteira não linear no espaço de entradas. (c) Fronteira linear no espaço de características.

Fonte: (LORENA, 2007)

como exemplo, os algoritmos ID3 (QUINLAN, 1986), C4.5 (QUINLAN, 1993) e o C5.0 (QUINLAN, 2010).

Uma árvore é uma estrutura de dados recursiva formada por um nó-folha que corresponde a uma classe ou por um nó de decisão que contém um teste sobre algum atributo. Para cada nó interno da árvore, existe uma aresta para uma subárvore. As subárvores possuem a mesma estrutura da árvore.

O método é bem simples e foi proposto por Hunt na década de 1960 (HUNT, 1966). A árvore é construída a partir de um conjunto T de casos de treinamento e assume-se que as classes são representadas por $(C_1, C_2, ..., C_k)$. O processo de construção da árvore de decisão é recursivo e baseia-se em múltiplas divisões que são feitas nos nós internos de acordo com algum critério de decisão baseado em seus valores. O algoritmo empregado em árvores de indução é um algoritmo recursivo de busca gulosa. Os passos para a construção da árvore são os seguintes (MONARD, 2003):

- T contém um ou mais exemplos, todos pertencentes à mesma classe C_j. Nesse caso, a árvore de decisão para T é um nó folha identificando a classe C_j;
- T não contém exemplos. Novamente, nessa situação, a árvore é uma folha, mas a classe associada à folha deve ser determinada a partir de informação além de T;
- T contém exemplos que pertencem a várias classes. Nesse caso a idéia é dividir T

em subconjuntos de exemplos que são (ou aparentam ser) conjuntos de exemplos pertencentes a uma única classe. Normalmente, um teste é escolhido baseado em um único atributo que possui resultados mutuamente exclusivos. Para isso, é escolhido um atributo preditivo A, que possui um ou mais possíveis resultados denotados por $(O_1, O_2, ..., O_n)$. T é particionado em subconjuntos $T_1, T_2, ..., T_n$, nos quais todos os T_i contém todos os exemplos em T que possuem como resultado daquele teste o valor O_i para o atributo A. A árvore de decisão para T consiste de um nó de decisão identificando o teste sobre o atributo A, e uma aresta para cada possível resultado, ou seja, n arestas.

- O algoritmo é repetido recursivamente (os passos 1, 2 e 3) para cada subconjunto de exemplos de treinamento de maneira que, em cada nó, as arestas levam para as subárvores construídas a partir do subconjunto de exemplos T_i.

- Após a construção da árvore de decisão, a poda pode ser realizada para melhorar a capacidade de generalização da árvore de decisão.

Sobre um nó da árvore, ocorre uma decisão, resultando no seu particionamento e a classificação de um exemplo decorre das decisões que foram tomadas de acordo com os valores dos atributos. O particionamento da árvore depende do critério de avaliação adotado para escolher o atributo. São exemplos de critérios:

- Ganho Máximo: é uma medida baseada na "impureza" dos dados. A medida desta impureza é chamada de entropia da informação. A entropia da informação tem origem em um trabalho de Shanon, Uma Teoria Matemática da Comunicação (SHANNON, 1948), que buscava quantificar a informação e cuja aplicação original era o campo das telecomunicações. O cálculo da entropia pode ser empregado para calcular e encontrar o atributo mais informativo para o nó de decisão na árvore. Quanto menor for a entropia, melhor será. A entropia de uma variável que possui c valores com probabilidades $p_1, p_2, ..., p_n$ é dada por:

$$Entropia(no) = \sum_{i=1}^{c} -p_i * log_2 p_i$$

onde p_i é a probabilidade ou fração dos registros que pertencem á Classe C_i e c é o número de classes.

Para determinar o quão boa é uma condição de teste realizada é necessário comparar o grau de entropia do nó-pai (antes da divisão) com o grau de entropia dos nós filhos (após a divisão). O atributo que gerar uma maior diferença é escolhido como condição de teste. O ganho da informação é obtido pela equação:

$$GanhoInformacao(GI) = Entropia(pai) - \sum_{i=1}^{n}(\frac{N(v_j)}{N} * entropia(v_j))$$

onde n é o número de valores do atributo, ou seja, o número de nós filhos, N é o número total de exemplos do nó-pai e $N(v_j)$ é o número de exemplos associados ao nó-filho v_j.

O critério de ganho seleciona como atributo-teste aquele que maximiza o ganho de informação. Sua grande desvantagem é que o ganho de informação dá preferência a atributos com muitos valores possíveis (número de arestas).

Quando isto ocorre, o valor da entropia é mínima porque, em cada nó, todos os exemplos (no caso um só) pertencem à mesma classe. Essa divisão geraria um ganho máximo, mas seria totalmente improdutivo.

- Razão de Ganho: Quinlan (QUINLAN, 1993) propôs o cálculo de razão de ganho para solucionar o problema do ganho da informação. A equação da razão de ganho é a seguinte:

$$RazaoGanho = \frac{GI}{Entropia(no)}$$

- Gini: para um problema de c classes, o índice de pureza Gini (CERIANI, 2012) pode ser calculado como:

$$gini(D) = 1 - \sum_{j=1}^{c} * p_j^2$$

onde: D é o conjunto de exemplos, c são as classes e p_j é a frequência relativa da classe j em D.

Se o valor mínimo de Gini for igual a 0, então os dados são "puros", sendo esta a melhor situação, pois todos os registros pertencem a uma mesma classe. Por exemplo, um conjunto de exemplos possui 6 instâncias. Estas instâncias devem ser classificadas em duas classes distintas 0 e 1. Vamos supor que o processo de indução, utilizando o índice Gini sobre um atributo, faz com que todos os exemplos sejam classificados na classe 0. Então o índice Gini seria igual a 0 e este seria o melhor atributo para realizar a classificação. Se o valor máximo de Gini é igual a (1 - 1/c), então todos os registros estão igualmente distribuídos entre todas as classes e esta é a pior situação. Assim, por exemplo, em um problema de classificação para divisão dos exemplos em duas classes, a pior situação seria $(1 - 1/2) = 0,5$. O atributo com este índice, para um problema com duas classes, não teria boa capacidade de generalização.

São exemplos de algoritmos de aprendizado de máquina que empregam o modelo de indução em árvores de decisão: o ID3, o C4.5 e o C5.0.

3.2.1 ID3

O ID3 *(Iterative Dichotomizer 3)* (QUINLAN, 1986) é um algoritmo recursivo baseado em busca gulosa sobre um conjunto de atributos para encontrar aqueles que melhor dividem os exemplos, gerando subárvores. Foi desenvolvido por Quinlan (QUINLAN, 1986), sendo o antecessor dos algoritmos C4.5 e C5.0. A principal limitação do ID3, segundo (VON ZUBEN), é que ele só lida com atributos categóricos não ordinais, não sendo possível apresentar a ele conjuntos de dados com atributos contínuos. Nesse caso, os atributos contínuos devem ser previamente discretizados. 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. 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 ter como consequência árvores mais complexas. O ID3 também não apresenta nenhum método de pós-poda, o que poderia amenizar esse problema de árvores mais complexas.

3.2.2 C4.5

O C4.5 (QUINLAN, 1993) é um algoritmo do tipo guloso pois, 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. Utiliza a estratégia "dividir para conquistar": partindo da raiz, criam-se subárvores até chegar às folhas, o que implica em uma divisão hierárquica em múltiplos subproblemas de decisão, os quais tendem a ser mais simples que o problema original.

O algoritmo constrói a árvore de decisão a partir de um conjunto de dados $S = s_1, s_2, ..,$ em que s_i consiste em um vetor p-dimensional $(x_1, i, x_2, i, ..., X_P, i)$, sendo que x_j representa um atributo ou característica de um exemplo i. Para cada nó, o C4.5 escolhe um atributo que divide mais efetivamente o conjunto de amostra em subconjuntos. O critério de divisão é a razão de ganho. Assim, ele considera apenas os atributos que possuem ganho da informação acima da média e escolhe o atributo com a maior razão de ganho para ser o raiz da árvore. As subárvores são construídas com os atributos restantes, recursivamente, utilizando o mesmo processo.

As melhorias do C4.5 em relação ao ID3 são (VON ZUBEN): manipulação de atributos contínuos: o algoritmo 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); permite que atributos com valores desconhecidos sejam tratados de forma especial e representados como '?'; utiliza a razão de ganho para selecionar o atributo que melhor divide os exemplos, gerando árvores mais precisas e menos complexas; consegue tratar problemas em que os atributos possuem custos diferenciados; apresenta um método de pós-poda das árvores geradas. O algoritmo faz uma busca na árvore, de baixo para cima, e transforma em nós folha aqueles ramos que não apresentam nenhum ganho significativo.

3.2.3 C5.0

O C5.0 (QUINLAN, 2010) é uma evolução do conhecido e muito utilizado algoritmo C4.5. O C5.0 realiza a classificação de um dado, utilizando um classificador que é representado

como uma árvore de decisão ou um conjunto de regras. O C5.0 divide a amostra tendo como base o atributo que resulta em maior ganho de informação em cada nível da árvore. O objetivo final é categorizar os exemplos da amostra. Múltiplas divisões em mais de dois subgrupos é permitido. O C5.0 constrói árvores de decisão em duas etapas: inicialmente uma grande árvore é construída, em seguida, é realizada a poda nessa árvore, ou seja, partes que classificam com um erro relativamente alto são removidas, sendo substituídas por folhas. O algoritmo de construção de árvore de decisão possui um mecanismo para selecionar uma amostra aleatória do conjunto de dados para ser usada na criação da árvore de decisão. Após isso, o classificador criado é testado com um conjunto disjunto de casos. São exemplos de características que foram incluídas no C5.0 (BUJLOW, 2012):

- O C5.0 pode incluir o *adaptative boosting* (SCHAPIRE, 2012) que consiste em gerar vários classificadores ao invés de apenas um. Quando um novo caso precisa ser classificado, este é classificado por cada classificador existente, sendo a classe final a classe em que o caso mais foi classificado (a escolha final resulta da votação dos classificadores). Outra melhoria criada foi a mineração dos atributos. Essa propriedade consiste em predefinir um conjunto de atributos que serão usados para construir uma árvore de decisão ou um conjunto de regras, melhorando a eficiência do algoritmo quando vários atributos apresentam grande ganho de informação, constituindo um grande conjunto de alternativas para cada nó.

- O C5.0 permite realizar a validação cruzada de dados. Esta validação consiste em unir todos os casos existentes nos arquivos de dados de amostra e de teste, e dividir esse conjunto em *n blocos*. Para cada bloco de dados B, um classificador é construído a partir dos casos existentes nos blocos restantes e é testado nesse bloco B. A taxa de erro de um classificador produzido dessa forma é calculada como a razão entre o número total de erros nos casos do bloco B e o número total de casos.

- O custo variável de classificação incorreta permite diferenciar os tipos de erros cometidos. Para cada par (classe prevista, classe real) é associado um custo, que é o valor que diferencia os tipos de erros. O mecanismo de atribuir um peso para cada caso é uma melhoria acrescentada ao C5.0 em relação ao C4.5. Esse mecanismo permite definir casos mais importantes dentro de um conjunto de dados. O uso dessa propriedade não assegura o aumento da precisão do classificador. A precisão da previsão somente será melhorada quando pesos atribuídos aos nós das árvores de

decisão possuírem valores similares, ou seja, quando casos de importâncias similares forem similares.

- Capacidade de tratar vários tipos de dados: no C5.0 foram adicionados vários tipos de dados que incluem data, tempo, *timestamp*, atributos discretos e rótulos, que não existiam no C4.5. Para atributos ausentes, o C5.0 permite que eles sejam tratados como não aplicáveis. Fornece também recursos que permitem definir novos atributos em função de outros atributos.

3.3 ALGORITMOS BASEADOS EM COMITÊS DE DECISÃO

Em aprendizado de máquina, pode-se empregar um comitê de métodos de aprendizado, todos produzindo classificadores para o mesmo problema e tomando como resultado final a votação destes classificadores para a atribuição da classe para uma instância do conjunto de exemplos.

O algoritmo da figura 3.4 apresenta um método geral utilizado por comitês.

FIG. 3.4: Comitê de aprendizado

1: $D \leftarrow$ Conjunto de Treinamento;
2: **para** $i \leftarrow 1$ **até** n **faça**
3: Criar(D_i a partir de D);
4: ConstruirClassificador (C_i a partir de D_i);
5: **fim para**
6: **para todo** $x \in T$ **faça**
7: $C^*(x) \leftarrow Voto(C_1(x), C_2(x)...C_n(x))$;
8: **fim para**

São exemplos de métodos de comitês utilizados em aprendizado de máquina (WITTEN, 2011): *bootstrap aggregating (bagging), boosting e stacking.*

- Bootstrap Agregating (Bagging): neste caso, são produzidas várias estruturas de decisão diferentes. Vamos supor que utilizamos o algoritmo Random Forest para produzir árvores de decisão. O comitê será formado por árvores de decisão que possuem poucas diferenças. Isto pode ser feito utilizando conjuntos de amostragem de mesmo tamanho e formando-os a partir do conjunto de exemplos. Dizemos que esta amostragem é construída com "substituição", significando que nas amostragens podem existir repetições de instâncias já inseridas em outras amostras. São produzidos diferentes conjuntos de treinamento, constituídos por exemplos escolhidos

aleatoriamente, e para cada um destes é construído um modelo. Após a classificação de cada um dos indutores ocorre a votação, sendo a classe mais votada a escolhida para o exemplo. Quando o aprendizado é por regressão a predição resultante é a média dos resultados dos classificadores.

- Boosting: neste caso, novos modelos são influenciados pelos resultados dos modelos anteriormente construídos. Vamos supor que criamos um modelo e observamos as instâncias e como elas foram classificadas, verificando a taxa de erros. Pela taxa de erros, concluímos que a classificação não foi a ideal e que tivemos uma dificuldade em classificar as instâncias, ou seja, nosso classificador é fraco. O *boosting* é uma técnica de aprendizado de máquina que combina diversos classificadores fracos com o objetivo de melhorar a acurácia geral. Em cada iteração, o algoritmo atualiza os pesos dos exemplos e constrói um classificador adicional, utilizando uma diferente versão do conjunto de dados de treinamento que é obtida pela variação do peso atribuído a cada exemplo (DUARTE, 2009). Os pesos maiores são atribuídos para os exemplos que são classificados incorretamente e menores para os que são classificados corretamente. Os exemplos com maiores pesos terão maior chance de serem escolhidos para o próximo conjunto de exemplos da próxima iteração. Um exemplo deste algoritmo é o Adaptative Boosting (AdaBoost).

- Stacking: este é um método que utiliza classificadores heterogêneos (CAFFÉ). O método utiliza uma estrutura em duas camadas. Na camada de nível 0, vários algoritmos de aprendizado recebem o conjunto de treinamento, gerando os classificadores de nível-0. Esta camada, processa os dados que servem de entrada para a próxima camada, o nível-1, que realiza combinações dos dados de entrada gerados pelos classificadores de nível 0 e gera o metaclassificador final. Neste método não é utilizada a votação de classificadores e podem ser utilizados diferentes algoritmos de aprendizado de máquina em sua execução.

3.3.1 RANDOM FOREST

O algoritmo Random Forest (BREIMAN, 2001) é um classificador baseado em árvores de decisão e pode reconhecer os padrões de várias classes ao mesmo tempo. O algoritmo é do tipo comitê e é composto por inúmeras árvores, formando assim florestas de decisão.

O algoritmo foi inicialmente proposto por (HO, 1995) e posteriormente desenvolvido por (BREIMAN, 2001).

O método combina o conceito de Bagging (BREIMAN, 1996) e de seleção aleatória de características para a construção de um conjunto de árvores com variância controlada.

Cada árvore dá um voto que indica sua decisão sobre a classe que pertencerá um determinado objeto. A classe com o maior número de votos é escolhida para o objeto.

As árvores individuais da floresta são construídas da seguinte forma:

- O número de casos no conjunto de treinamento é N. N casos aleatórios, mas obtidos com substituição, a partir do conjunto de exemplos original. N será a amostra para o treinamento das árvores da floresta.

- Se existem K variáveis de entrada, um número $k < K$ deve ser especificado de tal forma que para cada nó, k variáveis são selecionadas aleatoriamente a partir de K e a melhor divisão, com base no critério de ganho da informação ou índice Gini, sobre estas k variáveis é utilizado para dividir o nó;

- Toda árvore na floresta pode crescer até a máxima extensão possível. Não existe poda.

Podemos formalizar esta descrição conforme o algoritmo 3.5. Este algoritmo apresenta a técnica de construção de um comitê de árvores de decisão utilizando *bagging*.

A função $treinaDT(T', k)$ executa o treinamento de uma árvore de decisão sobre a amostra T' obtida pela execução da função $bootstrap(T)$ (*bagging*). k são as características selecionadas aleatoriamente em cada divisão. Após o treinamento, a árvore é adicionada ao comitê pela função $insereArvore(Tree, RF)$. No final do algoritmo a classificação é realizada por votação dos classificadores.

O processo de treinamento da árvore de decisão é descrito no algoritmo 3.6. O algoritmo é recursivo. Para cada divisão são selecionados k atributos aleatoriamente entre o total de K atributos. Para realizar a divisão do nó utiliza-se um critério de escolha como, por exemplo, o ganho da informação ou o índice Gini. O crescimento da árvore também pode ser limitado, evitando-se assim, a utilização de poda, e, consequentemente, haverá maior diversidade de árvores e redução de correlação. Quando não houver mais nós para serem inseridos na árvore ou quando o limite de profundidade for atingido, o algoritmo retornará a classe.

FIG. 3.5: Random Forest

1: $T \leftarrow (x, y)$
2: $m \leftarrow$ número de árvores (classificadores);
3: $k \leftarrow$ número de atributos aleatórios para cada árvore;
4: $K \leftarrow$ número total de atributos;
5: **para** $i \leftarrow 1$ **até** m **faça**
6: $\quad T' \leftarrow Bootstrap(\mathrm{T})$;
7: $\quad Tree \leftarrow treinaDT(T', k)$;
8: $\quad insereArvore(Tree, RF)$;
9: **fim para**
10: $Classificacao(Max(Votos(RF))$;

FIG. 3.6: Árvore de Decisão

1: $T \leftarrow (x, y)$;
2: $K \leftarrow$ número total de atributos;
3: $k \leftarrow$ número de atributos aleatórios k;
4: $lTamanho \leftarrow$ nível maximo do nó folha;
5: **se** $TamanhoDe(T') >= lTamanhoouFim(T')$ **então**
6: $\quad$ **retorne** $classe \leftarrow indexOf(max(histograma(T')))$
7: **senão**
8: $\quad atributos \leftarrow random(k, K)$;
9: $\quad atributo \leftarrow max(GanhoInformacao(atributos))$;
10: $\quad limite \leftarrow lTamanho$;
11: $\quad Divisao(T', atributo, limite)$;
12: $\quad AdNoEsquerda \leftarrow treinaDT(esquerdaT', k)$;
13: $\quad AdNoDireita \leftarrow treinaDT(direitaT', k)$;
14: **fim se**

O algoritmo Random Forest possui dois parâmetros importantes que afetam o seu desempenho consideravelmente: o número de árvores m do comitê e o número de atributos k que serão escolhidos aleatoriamente para cada árvore. Esta sua característica é importante e o diferencia de muitos outros algoritmos de comitê que necessitam de melhorias em vários parâmetros para funcionar corretamente.

O Random Forest, conforme implementado originalmente por Breiman, não necessita de uma amostra de teste e de validação cruzada pois, quando amostra os dados utilizando *bootstrap*, ele assegura que cada árvore na floresta possuirá cerca de 2/3 dos dados disponíveis e os 1/3 restantes serão utilizados para teste (dados $out - of - bag$).

O Out-of-Bag (OOB) é utilizado como o conjunto de teste para as árvores e permite estimar, sem viés, a taxa de erros do classificador.

A observação da taxa de erro de *out-of-bag* possibilita escolher melhor a quantidade de k atributos que serão escolhidos aleatoriamente. Sendo K o número total de atributos, um valor inicial para k é dado por $k = \sqrt{(K)}$ ou $log_2(K) + 1$. O valor de k pode ser ajustado até que seu valor resulte no menor erro estimado.

Quanto à escolha do número de árvores, trata-se de uma questão ainda aberta e recomenda-se que seja utilizado um grande número, por exemplo, 100 árvores e que esta quantidade seja ajustada pela observação da taxa de erros OOB.

Depois da construção das árvores, as proximidades podem ser calculadas para cada par de casos. Se dois casos ocupam um mesmo nó folha a proximidade é incrementada. Ao final da execução, as proximidades são normalizadas dividindo-as pelo número de árvores da floresta. As proximidades são úteis para identificar casos discrepantes e de agrupamentos de casos.

Por não utilizar de forma integral a coleção de dados formada, o algoritmo Random Forest é bastante rápido, tanto na fase de treinamento quanto na tomada de decisão. Além disso, a sua taxa de acerto é bastante elevada independentemente do tamanho da base de dados considerada, sendo que a sua taxa de erros depende, principalmente de dois fatores (BREIMAN, 2001):

- da correlação entre duas árvores em uma mesma floresta, sendo que, aumentando-se a correlação entre as duas aumenta-se também a taxa de erros; e,
- do peso que cada árvore individual representa na floresta, sendo que um classificador forte diminui a taxa de erros e um classificador mais fraco aumenta essa taxa.

O valor k retirado do vetor de características determina a eficiência do algoritmo. Conforme k diminui, diminui a correlação, mas diminui também a capacidade preditiva da árvore, e quando k aumenta, aumenta a força do classificador, mas aumenta também a correlação entre as árvores.

Para classificar um novo objeto a partir de um vetor de características, é computada a decisão de cada árvore de uma dada floresta e aquela classe que receber mais votos favoráveis é a classe atribuída ao exemplo sendo reconhecido: assumindo que $C_b(x)$ seja a predição de classe do *b-ésimo* classificador da floresta. Então,

$$C_{rf}^{B}(x) = maioria\_dos\_votos\{C_b(x)\}_1^B$$

.

Em problemas de regressão utiliza-se a média das predições obtidas:

$$f_{rf}^{B} = \frac{1}{B}\sum_{b=1}^{B} T_b(x)$$

3.4 VALIDAÇÃO E AVALIAÇÃO DO APRENDIZADO DE MÁQUINA

Nesta seção são apresentados conceitos, definições e métricas que normalmente são utilizados para se verificar a qualidade e o desempenho da hipótese.

O fenômeno de ***overfitting*** ocorre quando a hipótese se ajusta em excesso ao conjunto de exemplos de treinamento. Um modelo com *overfitting* deve ser evitado porque provavelmente ele não será capaz de representar a realidade. Como o conjunto de treinamento é apenas uma amostra de todos os exemplos do domínio, é possível induzir hipóteses que melhorem o seu desempenho no conjunto de treinamento, enquanto pioram o desempenho em exemplos diferentes daqueles pertencentes ao conjunto de treinamento. Nesta situação, o erro em um conjunto de teste independente evidencia um desempenho ruim da hipótese (MONARD, 2003).

O fenômeno de ***underfitting*** ocorre quando a hipótese se ajusta muito pouco ao conjunto de exemplos de treinamento. Isto pode ocorrer devido ao conjunto conter poucos exemplos representativos ou a um classificador subdimensionado ou a um alto fator de poda ou a uma combinação de todos estes fatores. Quando isto ocorre, pode-se observar que a melhora de desempenho, tanto do conjunto de treinamento quanto o de teste, sempre será muito pequeno.

Os valores dos atributos e rótulos de classe podem possuir imperfeições. A estas imperfeições chamamos de **ruído**. O ruído pode ter origem no processo que gerou os dados, no processo de transformação dos dados e nas classes rotuladas incorretamente.

Para tratar o ruído e o *overfitting* utiliza-se uma técnica de **poda**. O objetivo do método de poda é melhorar a taxa de acerto do modelo para os novos exemplos que ainda não foram utilizados no conjunto de treinamento. Existem dois tipos de poda: a pré-poda e a pós-poda. A **pré-poda** ocorre durante o processo de construção do modelo indutor. A **pós-poda** ocorre após a construção (HAN, 2006).

Uma hipótese pode ser avaliada a respeito de sua **completude**, se ela classifica todos os exemplos, e de sua **consistência**, se ela classifica corretamente todos os exemplos. Portanto, dada uma hipótese, ela pode ser (MONARD, 2003): completa e consistente; completa e inconsistente; incompleta e consistente; e, incompleta e inconsistente.

O **bias de aprendizado ou viés** é a diferença entre o valor do parâmetro e o valor produzido pelo indutor. Pode ser interpretado como a preferência de uma hipótese sobre outra, além da simples consistência com os exemplos. Um estimador ou uma regra de decisão com bias igual a zero é chamado de *unbiased* e seria a situação ideal. O *bias* está relacionado com a consistência dos estimadores. Uma sequência de estimadores é consistente se, e apenas se, eles convergem para um valor e o *bias* converge para zero.

O método de avaliação consistirá em estimar uma medida verdadeira (por exemplo: acurácia e taxa de erros). Para que esta avaliação seja confiável, os exemplos utilizados no treinamento do classificador devem ser diferentes dos exemplos utilizados para teste e ambos devem ser amostras representativas do problema de classificação. Para que dados sejam obtidos e avaliados, métodos de validação dos experimentos são adotados.

São exemplos de métodos de validação (MONARD, 2003):

- resubstituição: é um método que constrói o classificador e testa seu desempenho no mesmo conjunto de exemplos. Como consequência, sua estimativa é altamente otimista e superajustada ao conjunto de exemplos de treinamento. Quando isto ocorre, o *bias* do método é otimista e o desempenho ótimo do conjunto de treinamento não se estende aos conjuntos de teste. Portanto, este não é um método de avaliação justo.

- *holdout*: divide os exemplos em uma porcentagem fixa p para treinamento e $(1-p)$ para teste. Normalmente, utilizam-se as seguintes proporções: $p = 2/3$ e $(1-p) =$

1/3, não existindo fundamentos teóricos para esta escolha.

- amostragem aleatória: L hipóteses ($L < n$) são induzidas a partir de cada conjunto de treinamento. O erro final é a média dos erros de todas as hipóteses induzidas calculado em conjuntos de teste independentes e extraídos aleatoriamente.

- validação cruzada com k partições ou *k-fold cross-validation*: os exemplos são aleatoriamente divididos em k partições mutuamente exclusivas *(folds)* de tamanho aproximadamente igual a n/k exemplos. $(k-1)$ partições são usadas para treinamento e a hipótese é induzida com a partição que sobrou.

- validação cruzada estratificada ou *stratified cross-validation*: ao gerar as partições mutuamente exclusivas, a distribuição de classe (proporção de exemplos em cada uma das classes) é considerada. Assim, por exemplo, se o conjunto de exemplos possuir duas classes com distribuição 20% e 80%, cada partição também terá esta proporção.

- *Leave-one-out*: é um caso especial de validação cruzada. Para uma amostra de tamanho n, uma hipótese é induzida utilizando $(n-1)$ vezes e a hipótese é testada com o único exemplo remanescente. O processo se repete por n vezes e a cada vez uma hipótese é induzida deixando de considerar um único exemplo. O erro é a soma dos erros em cada teste dividido por n. O processo é computacionalmente dispendioso sendo utilizado somente quando as amostras são pequenas.

- *bootstrap*: o processo de classificação repete-se por diversas vezes. Valores como o *bias* e o erro são estimados a partir dos experimentos replicados. Cada experimento é conduzido com base em um novo conjunto de treinamento obtido por amostragem com reposição do conjunto original de exemplos.

Muitas métricas utilizadas na avaliação são construídas tendo como base uma matriz de confusão.

Uma **matriz de confusão** de uma hipótese oferece uma medida efetiva do modelo de classificação. Mostra o número de classificações corretas versus as classificações preditas para cada classe (WITTEN, 2011). A tabela 3.1 mostra a relação existente entre a classificação correta e a classificação predita. A partir dela várias métricas e conclusões podem ser tiradas do processo de classificação.

TAB. 3.1: Matriz de Confusão

MATRIZ DE CONFUSÃO PARA DUAS CLASSES			
		INDUTOR	
		POSITIVO	**NEGATIVO**
ESPECIALISTA	**POSITIVO**	**Verdadeiro Positivo**	**Falso Negativo**
	NEGATIVO	**Falso Positivo**	**Verdadeiro Negativo**

Em uma matriz de confusão, os **verdadeiros positivos (VP)** e os **verdadeiros negativos (VN)** são as classificações corretas. Neste caso a classificação inicial (real, feita pelo especialista) e a predição (feita pelo aprendizado de máquina) estão de acordo. Um **falso positivo (FP)** ocorre quando a predição está incorreta como sim (positivo) pois deveria ser não, como realmente classificado e um **falso negativo (FN)** ocorre quando a predição está incorreta como não (negativo) pois deveria ser sim como realmente classificado. Exemplo: para o caso do verdadeiro positivo, o classificador real e a predição concordam que o arquivo é um *malware*; para o caso do verdadeiro negativo, o classificador real e a predição concordam que o arquivo não é um *malware* (benigno); para o caso falso negativo (FN) o classificador real define o arquivo como *malware* (sim, é maligno) e o classificador prediz o arquivo como benigno (não, o arquivo é benigno); para o caso do falso positivo (FP), o classificador real classifica que o arquivo não é um *malware* (não, o arquivo é benigno) e o classificador prediz a classe como *malware* (sim, o arquivo é um *malware*).

A **acurácia** é o número de verdadeiros positivos e verdadeiros negativos dividido pelo número total de exemplos.

$$Acc = \frac{VP + VN}{VP + FP + VN + FN}$$

A **taxa de erros** é o complemento da acurácia ou a soma dos falsos positivos e falsos negativos dividido pelo número total de exemplos.

$$TErr = 1 - Acc$$

ou

$$TErr = \frac{FP + FN}{VP + FP + VN + FN}$$

A **taxa de verdadeiros positivos (TVP)**, **revocação** ou **abrangência** é a razão entre os verdadeiros positivos (VP) e o número total de positivos. É a proporção de exemplos positivos que foram corretamente classificados. Exemplo: a probabilidade de o teste fornecer um resultado positivo (*malware*) dado que o arquivo é realmente um *malware*.

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

A **taxa de falsos positivos (TFP)** é a razão entre os falsos positivos (FP) e o número total de negativos. É a proporção de exemplos positivos que foram incorretamente classificados como positivos. Exemplo: a probabilidade de o teste fornecer um resultado positivo (o arquivo é um *malware*) dado que o arquivo não é na realidade um *malware*.

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

ou

$$TFP = 1 - ESP$$

A **especificidade** (ESP) ou **taxa de verdadeiros negativos** é o resultado da divisão entre os verdadeiros negativos (VN) e os que são realmente negativos (VN + FP).

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

ou

$$ESP = 1 - TFP$$

A **precisão** é a divisão entre os verdadeiros positivos (VP) e aqueles que foram preditos como positivos (VP + FP). É a capacidade do classificador em reconhecer as instâncias de uma classe de interesse e rejeitar as demais, ou seja, é a capacidade do indutor de reconhecer que o arquivo é um *malware*.

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

A **média harmônica de precisão e revocação** ou **F-measure** é a média harmônica entre a precisão e a revocação. F-measure será alto somente se a precisão e revocação também forem altos. O valor de F-measure está no intervalo entre 0 e 1 e a média harmônica entre dois números x e y tende a ser próxima de $min(x, y)$. Valores maiores de *F-measure* indicam uma melhor qualidade da classificação.

$$Fmeasure = \frac{2}{\frac{1}{PREC} + \frac{1}{TVP}} = \frac{2 * PREC * TVP}{PREC + TVP}$$

O cenário ideal da relação entre precisão e revocação seria aquela em que a precisão e a revocação fossem iguais a 1. Neste cenário, *F-measure* seria igual a 1 e o classificador seria perfeito. Nesta situação ideal, o número de falsos positivos e falsos negativos seria 0, ocorrendo total concordância entre a análise feita pelo especialista e a predição feita pelo classificador.

Uma métrica que indica o grau de concordância entre dois classificadores é a Estatística Kappa. O coeficiente Kappa (COHEN, 1960) é uma medida estatística que leva em consideração as probabilidades de as concordâncias de itens categóricos terem acontecido ao acaso, conforme a fórmula:

$$\kappa = \frac{Pr(a) - Pr(e)}{1 - Pr(e)}$$

onde k é o valor do índice kappa resultante, $Pr(a)$ é a proporção de vezes que os classificadores (neste caso, o especialista e a classificação resultante do aprendizado de máquina) concordaram ou é a concordância relativa observada e $Pr(e)$ é a proporção de vezes em que os k classificadores esperam concordar ao acaso ou é a probabilidade de ocorrer a concordância da hipótese ao acaso.

$$Pr(a) = Acc = \frac{VP + VN}{VP + VN + FP + FN} = \frac{VP + VN}{TOTAL}$$

e,

$$Pr(e) = P(A \cap A') + P(B \cap B') =$$
$$((\frac{VP + FN}{TOTAL} * \frac{VP + FP}{TOTAL}) + (\frac{VN + FP}{TOTAL} * \frac{VN + FN}{TOTAL}))$$

Uma concordância perfeita corresponde ao índice $kappa = 1$ e a falta de concordância (e.g. a ocorrência das concordâncias puramente ao acaso, ao índice $kappa = 0$).

A semântica da concordância Kappa é apresentada na tabela 3.2:

TAB. 3.2: Índice de concordância Kappa

K	**Interpretação**
< 0	Nenhuma concordância
0 a 0,2	Leve concordância
0,21 a 0,4	Concordância regular
0,41 a 0,6	Concordância moderada
0,61 a 0,8	Concordância substancial
0,81 a 1	Concordância quase perfeita

Índices de concordância Kappa acima de 0.60 são considerados satisfatórios em trabalhos científicos (LANDIS, 1977).

3.5 FRAMEWORK DE APRENDIZADO DE MÁQUINA

Um *framework* é formado por um conjunto de aplicações que possuem uma funcionalidade comum, ou seja, as aplicações devem pertencer a um mesmo domínio de problema. Um *framework* de aprendizado de máquina reúne vários algoritmos que permitem ao computador aprender tendo como base dados empíricos. Alguns exemplos de *frameworks open-source* de aprendizado de máquina:

- Weka: *framework* popular de aprendizado de máquina escrito em Java e desenvolvido pela Universidade de Waikato (http://www.cs.waikato.ac.nz/ml/weka/).
- Apache Mahout: *framework* constituído por algoritmos implementados para uso nas áreas de filtragem colaborativa, *clustering* e classificação. Muitas das implementações utilizam a plataforma Apache Hadoop (https://mahout.apache.org/).
- FAMA: *framework* de propósito geral para tarefas de aprendizado de máquina. Provê as interfaces genéricas para o desenvolvimento colaborativo de algoritmos e classificadores de aprendizado de máquina (https://code.google.com/p/fama/).
- mlpack: biblioteca de aprendizado de máquina desenvolvida em C++ (http://www.mlpack.org/about.html).
- Shark: biblioteca de aprendizado de máquina desenvolvida em C++ (http://image.diku.dk/shark/sphinx_pages/build/html/index.html).
- Dlib C++: conjunto de bibliotecas de aprendizado de máquina desenvolvidas em C++ (http://dlib.net/ml.html).
- PyBrain: biblioteca modular de aprendizado de máquina em Python (http://pybrain.org/).
- PyML: *framework* para aprendizado de máquina orientado a objetos escrito em Python (http://pyml.sourceforge.net/).
- Scikit-learn: ferramentas para mineração e análise de dados desenvolvidas em Python (http://scikit-learn.org/stable/).
- JSAT: *framework* constituído de vários algoritmos de aprendizado de máquina para classificação, regressão e *clustering* (https://github.com/josephmisiti/awesome-machine-learning#java-general-purpose).

- H2O: conjunto de programas que suportam aprendizado de máquina sobre dados armazenados em HDFS (https://github.com/0xdata/h2o).

3.5.1 FAMA

O FAMA (Framework de Aprendizado de Máquina) (FAMA, 2014) consiste de um conjunto de programas que implementam algoritmos para aprendizado de máquina e foi desenvolvido com a linguagem de programação C++.

A figura 3.7 mostra as principais classes que formam o núcleo do *framework*: *Validador*, *Avaliador*, *Treinador*, *Corpus* e *Classificador*.

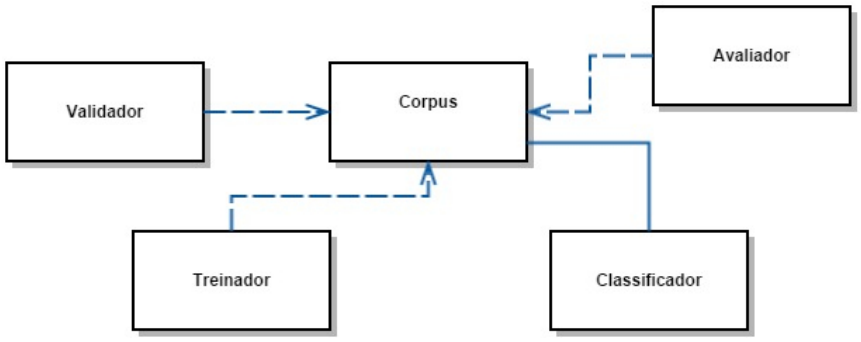


FIG. 3.7: Diagrama de Classes do FAMA

Corpus é a classe que armazena as informações na qual os métodos de aprendizado de máquina são aplicados. Para acessá-la, a classe Corpus disponibiliza dois métodos: *carregarArquivo* e *gravarArquivo*, que servem para carregar os arquivos no formato texto para a memória principal e para gravar os dados na memória em um arquivo texto, respectivamente. Os dados que são carregados na memória, por um objeto da classe herdeira de Corpus, é passado para a classe Treinador.

Treinador é a classe utilizada pelo processo de aprendizagem do algoritmo. A classe Treinador tem o método abstrato *executarTreinamento* que é implementado pelas classes derivadas dele, cada uma contendo um método diferente de aprendizado. Exemplo: estes métodos são implementados em alguns algoritmos de aprendizado de máquina como o SVM, ID3, HMM, Decision Stump, Random Forest, regressão linear, regressão logística e C5.0.

Classificador é a classe que utiliza os resultados de *Treinador* para classificar os exemplos. O método *executarClassificação* é implementado pelas classes derivadas e define as classes que serão atribuídas aos exemplos resultantes do processo de treinamento.

Validador é a classe utilizada para implementar métodos aplicados no processo de validação do aprendizado. Para executar o método de validação cruzada com k-partições é executado, por exemplo, o método *executarExperimento* pelo ValidadorKDobras.

Avaliador é a classe utilizada para determinar o desempenho obtido pelo classificador. O método utilizado neste processo é o *calcularDesempenho*. Utilizado, por exemplo, para calcular a matriz de confusão e a acurácia do experimento.

4 TRABALHOS RELACIONADOS

O processo de análise manual de *malwares* por um especialista de segurança demanda tempo e, assim, sistemas podem ficar vulneráveis a um ataque de um código malicioso recém-criado. Além disto, *malwares* modernos implementam técnicas de evasão das análises estática e dinâmica de *malwares* com o objetivo de impedir a sua detecção.

Devido a estes problemas, vários trabalhos têm sido desenvolvidos na tentativa de aprimorar as técnicas de análise e identificação de *malwares*. A seguir, citamos alguns trabalhos na área e que se relacionam com nossa pesquisa:

A utilização da técnica de *sandbox*, para a realização da análise dinâmica de *malwares*, e o aprendizado de máquina são utilizados para automatizar a identificação de *softwares* maliciosos no trabalho de (DEANDRADE, 2013). O processo de coleta de dados foi baseado na análise dinâmica, que permite extrair informações dos processos executados por arquivos com formato Windows PE32.

Foram selecionados 13 atributos (características), cujos valores, para cada uma das instâncias do conjunto de exemplos, foram utilizados como entrada para o processo de aprendizado de máquina. Os algoritmos de aprendizado de máquina utilizados foram o Naive Bayes, Support Vector Machines (SVM), J48 e o Random Forest implementados no Weka. Neste trabalho foram realizados 4 experimentos e a tabela 4.1 resume a quantidade de artefatos utilizados na realização de cada um deles. O melhor resultado obtido nos experimentos foi o que utilizou a classificação com o Random Forest que teve acurácia igual a $93,6\%$.

(SAMI, 2010) apresenta um *framework* para análise e classificação de arquivos com formato PE, utilizando técnicas de mineração de dados. O *framework* tem como base o princípio de que APIs do Windows, utilizadas pelas aplicações que executam neste sistema, possuem características que podem ser extraídas e exploradas como um conhecimento sobre o comportamento dos executáveis. Utilizam em seus experimentos 32000 arquivos maliciosos, estando inclusos nesta coleção exemplos de Trojans, Backdoors, Spywares, Worms, etc e aproximadamente 3000 arquivos benignos, que incluem arquivos de sistema do Windows e aplicativos. O *framework* consta de três partes: o primeiro componente é um analisador PE que examina a estrutura do arquivo e extrai as APIs que foram

TAB. 4.1: Experimentos: análise automatizada de malwares

Exp	Nomenclatura	Qtd	Total
1	worms	4617	10000
	benignos	5383	
2	trojans	11115	20034
	benignos	8919	
3	backdoors	9591	18510
	benignos	8919	
4	worms	3000	17919
	trojans	3000	
	backdoors	3000	
	benignos	8919	

importadas pelo arquivo PE, sendo portanto, utilizado o método de análise estática de *malwares*; o segundo componente gera os atributos que devem ser analisados; o terceiro componente utiliza algoritmos de aprendizado de máquina para classificar os *malwares*. Foram utilizados nesta fase os algoritmos de aprendizado de máquina Random Forest, J48 e Naive Bayes, implementados no Weka. O melhor resultado obteve 98,3% de acurácia utilizando o algoritmo Random Forest.

Malicious Executable Classification System (MECS) (KOLTER, 2004) utiliza a análise estática de *malwares* para detectar arquivos executáveis, de qualquer formato, sem utilizar qualquer técnica ou sistema para remoção de ofuscação de código. Para a realização do experimento foram utilizados 1971 arquivos executáveis benignos e 1651 executáveis malignos. As características dos *malwares* foram extraídas de sequências de *bytes* do código executável desmontado e foram convertidos em *n-gramas*. Estes dados organizados foram submetidos a vários algoritmos de aprendizado de máquina: Instance Based K-nearest-neighboors (IBK) (WITTEN, 2011), Naive Bayes, Support Vector Machines (SVMs), Decision Trees (WITTEN, 2011), Boosted Naive Bayes (ELKAN, 1997) e Boosted Decision Trees (ROEA, 2005). Os melhores resultados obtidos foram os resultantes da classificação utilizando árvores de decisão. A taxa de verdadeiros positivos obtida foi igual a 98%

(RAMAN, 2012) utiliza a análise estática de *malwares* para obter os valores de sete características que são utilizadas para classificar um arquivo como benigno ou maligno. Para selecionar estes atributos são utilizadas diferentes características que estão presentes em partes de um arquivo Windows PE32. Estes dados organizados são utilizados como entrada de dados para os algoritmos de aprendizado de máquina. A base de dados foi formada por 5193 malwares e 3722 não malwares. O algoritmo de aprendizado de máquina

utilizado foi o Random Forest e a acurácia obtida foi igual a 92,34%.

(SIDDIQUI, 2008) utiliza a análise estática de *malwares* e um processo de mineração completo, desde a preparação dos dados até construção do modelo indutor. As características foram extraídas de sequências de instruções obtidas a partir da desmontagem do código e compreendem informações á nível sintático e semântico. Utilizou no experimento 2775 arquivos Windows PE32, sendo que 1444 arquivos eram malignos e 1330 eram arquivos benignos. Utilizaram para a classificação os algoritmos de aprendizado de máquina Random Forest, árvores de decisão e bagging. O melhor resultado obtido foi o do algoritmo Random Forest que alcançou 96% de acurácia.

(KONG, 2013) exploram técnicas que permitem automatizar a classificação de *malwares* em suas famílias correspondentes, utilizando a análise estática de *malwares*. Apresentam um *framework* genérico que extrai informações estruturais de *malwares* a partir de grafos de chamadas de APIs em que importantes características são codificadas como atributos á nível de funções. O *framework* permite avaliar a similaridade entre *malwares* com base no aprendizado de suas características. Para combinar os vários tipos de atributos de *malwares*, o método aprende adaptativamente o nível de confiança associado com a capacidade de classificação de cada tipo de atributo e então adota um classificador de comitê para automatizar a classificação. Para avaliar a abordagem, utilizam instâncias de *malwares* baseadas em Windows pertencentes a 11 famílias.

O sistema Chatter (MOHAISEN, 2014) utiliza a análise dinâmica e o aprendizado de máquina para predizer três famílias de *malwares*. Utilizam características de *malwares* extraídas de processos em execução pelo *malware*, interação do *malware* com registros do sistema, operações sobre sistemas de arquivos e interações de redes. Para reduzir a possibilidade de evasão da análise pelos *malwares* os autores propõem o sistema Chatter. Utilizaram 2069 exemplares de malwares classificados em três famílias. O sistema Chatter preocupa-se com a ordem que os eventos de sistema de alto nível ocorrem. Eventos individuais são mapeados em um alfabeto e traços de execução são capturados por meio de concisas concatenações das letras do alfabeto. Utilizam um corpus de *malwares* rotulado e técnicas de classificação de documentos utilizando n-gramas para produzir dados para um classificador que prediz a família de *malwares*. Os algoritmos de aprendizado de máquina utilizados foram o k-nearest neighboor (kNN), o SVM e árvores de decisão. O melhor índice de desempenho alcançado ocorreu com o kNN que obteve acurácia de 83,90%.

Neste trabalho utilizamos as técnicas de análise estática e dinâmica, unificando as

características que serão submetidas aos algoritmos de aprendizado de máquina e classificamos os códigos maliciosos por tipos de *malwares*.

5 UNIFICAÇÃO DAS ANÁLISES DE MALWARES

Este trabalho de pesquisa foi desenvolvido conforme as etapas apresentadas na figura 5.1. Na primeira etapa o ambiente de análise foi preparado. Esta etapa compreende a escolha, a instalação e configuração dos *softwares* necessários. Na segunda etapa as amostras de *malwares* e *não malwares* para compor os conjuntos de exemplos foram obtidas. Na terceira etapa foram escolhidas as características dos *malwares* que deveriam ser analisadas. Na quarta etapa foram executadas as análises estática e dinâmica dos *malwares* para obter os valores de suas características. Na quinta etapa os dados foram preparados para serem armazenados nos vetores de dados de entrada dos algoritmos de aprendizado de máquina. Na sexta etapa os algoritmos de aprendizado de máquina foram preparados e na sétima etapa os resultados da classificação dos *malwares* foram obtidos e analisados.

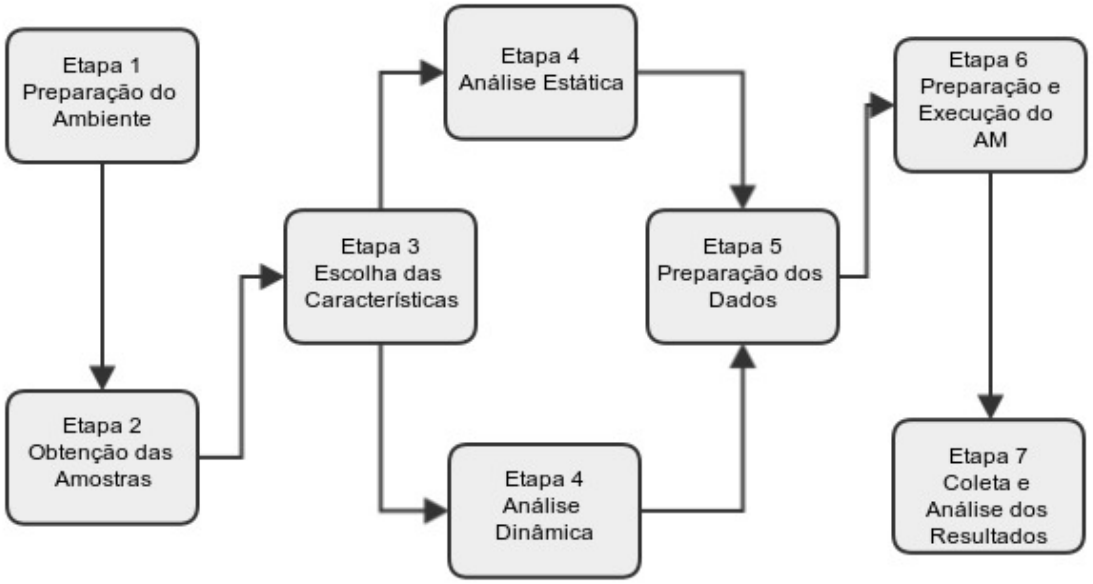


FIG. 5.1: Etapas da Unificação da Análise de Malwares

5.1 PREPARAÇÃO DO AMBIENTE DE ANÁLISE

Na primeira etapa foi preparado o ambiente de análise composto por uma máquina hospedeira *(host)* utilizada para realizar o gerenciamento das tarefas de análise e hospedar a máquina virtual com o sistema alvo da atuação dos arquivos de *malwares* e não *malwares*. A figura 5.2, mostra o esquema do ambiente que foi montado para o desenvolvimento deste trabalho.

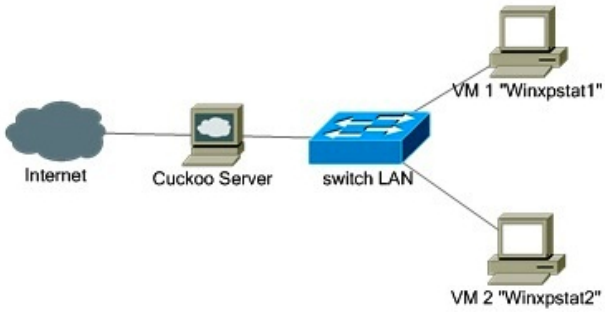


FIG. 5.2: Diagrama do ambiente de análise

A máquina hospedeira usada (Cuckoo Server) possui 4GB de memória RAM, capacidade para instalar duas máquinas virtuais e 500 GB de espaço de armazenamento secundário. Os *softwares* utilizados em todas as fases do trabalho são livres com exceção do sistema operacional Windows XP, instalado na máquina virtual, e que foi utilizado como sistema alvo da execução dos arquivos de *malwares* e não *malwares*. Também foi instalado o conjunto de programas de escritório Office 2007 e o programa Acrobat Reader, para o caso de utilização destes *softwares* por algum *malware*. A máquina hospedeira foi configurada com o sistema operacional Linux, Ubuntu, distribuição 14.04 LTS. O *software* de virtualização utilizado foi o VirtualBox 4.2.16 Ubuntu r86992.

Os *softwares* e aplicativos utilizados na análise estática foram o exiftool (HARVEI, 2014), pescanner.py (LIGH, 2010) e o programa Pyew.

Para a análise dinâmica foi utilizado o Cuckoo Sandbox versão 1.0. Antes de instalar e configurar o Cuckoo Sandbox foram instalados os pacotes do python e do SqlAlchemy, ambos obrigatórios para o funcionamento. Foram instalados outros aplicativos e bibliotecas que são opcionais, mas que permitem ajustar o sistema de acordo com a finalidade da análise pretendida. Os aplicativos e bibliotecas opcionais que foram instalados são os seguintes: dpkt, utilizado para extrair informações de arquivos Pcap; jinja2, utilizado para melhorar a apresentação dos relatórios gerados em html; magic, utilizado para iden-

tificar o formato dos arquivos; pydeep e ssdeep, utilizados para calcular o *fuzzy hash* dos arquivos; yara e yara python, utilizados para encontrar assinaturas equivalentes em arquivos; bottlepy, servidor web utilizado pelo Cuckoo para apresentação dos resultados; pefile, utilizado na análise estática de arquivos binários com formato PE32 e tcpdump, utilizado para captura e análise de tráfego de rede.

Para dificultar o emprego de técnicas anti-vm, configuramos o VirtualBox de tal forma a dificultar a utilização de sensores pelo *malware*. Para realizar esta configuração, baixamos o pacote CuckooMon a partir do repositório https://github.com/ cuckoobox/ cuckoomon. A figura 5.3 mostra os arquivos que foram baixados e que permitem configurar o CukooSandbox contra anti-vm.

```
reinaldo@reinaldo-Lenovo-IdeaPad-G485:~/cuckoomon$ ls
bson                hook_reg_native.c  logtbl.pyc
config.c            hook_services.c    lookup.c
config.h            hooks.h            lookup.h
cuckoomon.c         hook_sleep.c       Makefile
cuckoomon.dll       hook_sleep.h       Makefile~
distorm3.2-package  hook_socket.c      Makefile.orig
hook_file.c         hook_special.c     misc.c
hook_file.c~        hook_sync.c        misc.h
hook_file.c.orig    hook_thread.c      netlog.py
hook_file.h         hook_window.c      ntapi.h
hooking.c           ignore.c           objects
hooking.h           ignore.h           pipe.c
hook_misc.c         install.sh         pipe.h
hook_network.c      LICENSE.txt        README.md
hook_process.c      log.c              tests
hook_reg.c          log.h              utf8.c
hook_reg.c~         logtbl.c           utf8.h
hook_reg.c.orig     logtbl.py
```

FIG. 5.3: Arquivos do CuckooMon

No extrato do arquivo hook_reg.c, mostrado na figura 5.4 existe uma *hook function* que possui como parâmetro o valor RegOpenKeyExA. lpSubKey é utilizado para verificar as *strings* VirtualBox ou ControlSet.

Quando um *malware* que possui sensores tenta detectar a máquina virtual, buscando por estas *strings*, o código do programa detectará esta ação do *malware* e falsificará a resposta, enganando-o, dando-lhe uma resposta de que o sistema sobre o qual ele roda não é uma máquina virtual.

```
/* Hardened */

HOOKDEF(LONG, WINAPI, RegOpenKeyExA,
__in HKEY hKey,
__in_opt LPCTSTR lpSubKey,
__reserved DWORD ulOptions,
__in REGSAM samDesired,
__out PHKEY phkResult ) {

LONG ret;
if (strstr(lpSubKey, "VirtualBox") != NULL) {
 ret = 1;
 LOQ("s", "Hardening", "Faked RegOpenKeyExA return");
}

else if (strstr(lpSubKey, "ControlSet") != NULL) {
 ret = 1;
 LOQ("s", "Hardening", "Faked RegOpenKeyExA return");

}

else {
 ret = Old_RegOpenKeyExA(hKey, lpSubKey, ulOptions, samDesired, phkResult);
}
 LOQ("psP", "Registry", hKey, "SubKey", lpSubKey, "Handle", phkResult);
 return ret;
}
```

FIG. 5.4: Código de detecção de anti-vm

5.2 OBTENÇÃO DAS AMOSTRAS DE MALWARES E NÃO MALWARES

Na etapa 2 foram obtidas as amostras de *malwares* e não *malwares* utilizadas no experimento. Os *malwares* foram obtidos no site VirusShare (http://vírusshare.com). Foram baixados um total de 131.073 *malwares*. Os arquivos de *malwares* possuem diversos formatos, PE32, HTML, ico, gif, pdf, doc, xls, jpg, etc. Após a análise foram selecionados da população somente aqueles que possuiam o formato Windows PE32. Isto foi feito para manter uma padronização da análise e também devido ao volume de ataques conhecidos contra sistemas Windows. A coleção possui arquivos de *malwares* que foram obtidos até o primeiro bimestre de 2014.

Além do conjunto de *malwares*, também foram obtidas amostras de arquivos que classificamos como não *malwares* para fins de comparação com a base de *malwares* e futura submissão das características das duas populações aos algoritmos de aprendizado de máquina. Foram obtidos 2659 exemplares de arquivos não *malwares*. Estes arquivos foram obtidos dos *sites* sourceforge.net (http://www.sourceforge.net), oldapps (http://www.oldapps.com), de servidores de ftp anônimos (http://www.ftp-sites.org/) e de arquivos de nossos computadores.

5.3 ESCOLHA DE CARACTERÍSTICAS

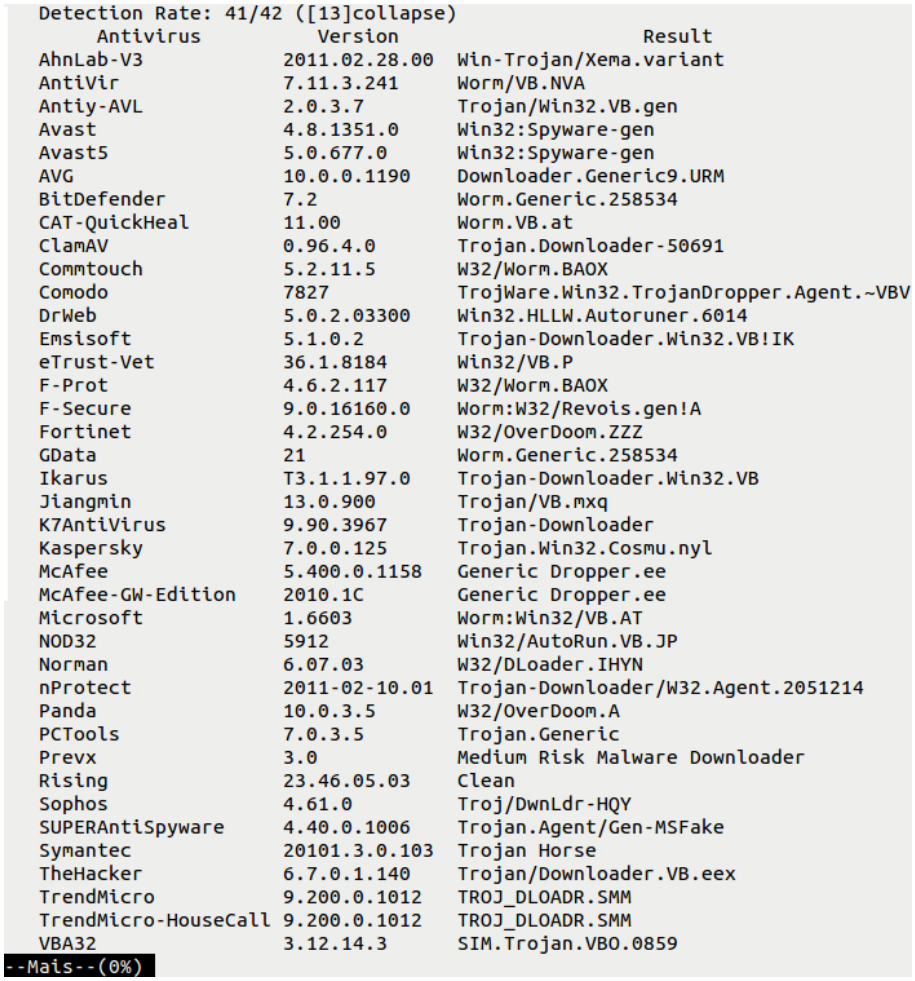
Na etapa 3 foram selecionadas as características para análise dos *malwares*. Além das escolhas das características, classificamos os arquivos como *malwares* e não *malwares* e também em tipos de *malwares*, seguindo a definição dada pelo (CERT.BR, 2013): *trojan, vírus, worm, backdoor, rootkit, bot e spyware*. A classe benigno foi também utilizada e representa os arquivos que não são *malwares*.

Para classificar os arquivos em tipos de *malwares* utilizamos o Virustotal. Todos os arquivos de *malwares* que foram obtidos na etapa 2, foram verificados no vírustotal e confirmados como *malwares* por pelo menos um antivírus. A figura 5.5 mostra o resultado da análise estática, a partir de um extrato do arquivo de análise de um *malware*. Na figura é mostrado o número de antivírus que identificaram o arquivo como *malware*. Neste caso, a taxa de detecção foi de 41 antivírus entre 42, que identificaram o arquivo como malicioso. Também podemos notar que não existe uma padronização na nomeação dos *malwares* e que ocorre uma discordância na classificação em relação aos tipos.

Para classificar os tipos de *malwares* criamos um dicionário com *strings* que foram utilizadas na busca de *strings* semelhantes na listagem de detecção dos antivírus. Utilizando estas *strings* do dicionário contamos as suas frequências na lista de detecção. A *string* com maior frequência foi utilizada para definir o tipo do *malware*.

As características ou atributos escolhidos para a análise estática são os seguintes:

- Yara: obtido pela execução do aplicativo yara.py, integrado ao Cuckoo Sandbox, e que permite verificar *malwares* que já são conhecidos e que já estão mapeados;
- EPEiD: verifica se existe alguma assinatura de código de cifradores, empacotadores e compiladores conhecidos e que foram utilizados na criação do código do *malware*.
- DateSusp: corresponde ao atributo de identificação de datas suspeitas. Datas de criação suspeitas foram obtidas como resultado da análise utilizando o aplicativo pescanner.py. A data é considerada suspeita se o ano for menor que 2000 ou maior que o ano atual. O cálculo é feito com base no *timestamp* do arquivo e indica quando o compilador produziu o arquivo. Quando a data for suspeita é atribuído o valor 1 para o atributo e quando for normal é atribuído 0.
- EPSusp: permite identificar se o ponto de entrada de uma seção PE é suspeita. Um ponto de entrada de uma seção é o nome de uma seção PE que contém o Address-



```
Detection Rate: 41/42 ([13]collapse)
     Antivirus          Version                    Result
AhnLab-V3            2011.02.28.00  Win-Trojan/Xema.variant
AntiVir              7.11.3.241     Worm/VB.NVA
Antiy-AVL            2.0.3.7        Trojan/Win32.VB.gen
Avast                4.8.1351.0     Win32:Spyware-gen
Avast5               5.0.677.0      Win32:Spyware-gen
AVG                  10.0.0.1190    Downloader.Generic9.URM
BitDefender          7.2            Worm.Generic.258534
CAT-QuickHeal        11.00          Worm.VB.at
ClamAV               0.96.4.0       Trojan.Downloader-50691
Commtouch            5.2.11.5       W32/Worm.BAOX
Comodo               7827           TrojWare.Win32.TrojanDropper.Agent.~VBV
DrWeb                5.0.2.03300    Win32.HLLW.Autoruner.6014
Emsisoft             5.1.0.2        Trojan-Downloader.Win32.VB!IK
eTrust-Vet           36.1.8184      Win32/VB.P
F-Prot               4.6.2.117      W32/Worm.BAOX
F-Secure             9.0.16160.0    Worm:W32/Revois.gen!A
Fortinet             4.2.254.0      W32/OverDoom.ZZZ
GData                21             Worm.Generic.258534
Ikarus               T3.1.1.97.0    Trojan-Downloader.Win32.VB
Jiangmin             13.0.900       Trojan/VB.mxq
K7AntiVirus          9.90.3967      Trojan-Downloader
Kaspersky            7.0.0.125      Trojan.Win32.Cosmu.nyl
McAfee               5.400.0.1158   Generic Dropper.ee
McAfee-GW-Edition    2010.1C        Generic Dropper.ee
Microsoft            1.6603         Worm:Win32/VB.AT
NOD32                5912           Win32/AutoRun.VB.JP
Norman               6.07.03        W32/DLoader.IHYN
nProtect             2011-02-10.01  Trojan-Downloader/W32.Agent.2051214
Panda                10.0.3.5       W32/OverDoom.A
PCTools              7.0.3.5        Trojan.Generic
Prevx                3.0            Medium Risk Malware Downloader
Rising               23.46.05.03    Clean
Sophos               4.61.0         Troj/DwnLdr-HQY
SUPERAntiSpyware     4.40.0.1006    Trojan.Agent/Gen-MSFake
Symantec             20101.3.0.103  Trojan Horse
TheHacker            6.7.0.1.140    Trojan/Downloader.VB.eex
TrendMicro           9.200.0.1012   TROJ_DLOADR.SMM
TrendMicro-HouseCall 9.200.0.1012   TROJ_DLOADR.SMM
VBA32                3.12.14.3      SIM.Trojan.VBO.0859
--Mais--(0%)
```

FIG. 5.5: Identificação de malware pela análise estática

ofEntryPoint. O AddressofEntryPoint para arquivos legítimos ou não empacotados normalmente reside em uma seção nomeada .code ou .text para programas em modo usuário e PAGE ou INIT para *drivers* de *kernel*. Uma seção PE é considerada suspeita se o endereço do ponto de entrada está localizado na última seção do arquivo PE ou se o endereço não é reconhecido como normal. Quando o ponto de estrada da seção for suspeito é atribuído o valor 1 para o atributo e quando for normal é atribuído 0.

- SuspEntropy: identifica seções que possuem uma entropia muito alta ou muito baixa. A entropia é um valor entre 0 e 8 e que identifica a aleatoriedade dos dados em um arquivo. Esta classificação deriva-se do trabalho de (LYDA, 2007) que sugere que um arquivo com uma entropia maior que 7 deve ser considerado como suspeito. (ZABIDI, 2012) utiliza a entropia entre 0 e 1 ou maior que 7 ($0 < y < 1 V y > 7$)

para identificar um arquivo suspeito. Por exemplo, um arquivo com uma sequência muito longa de caracteres iguais possui baixa entropia e dados criptografados ou compactados possuem alta entropia. O cálculo da entropia permite identificar seções que possuem código anormal ou empacotado. Quando a entropia for suspeita é atribuído o valor 1 para o atributo e quando for normal é atribuído 0.

- NumberOfSections: número de seções que compõem a imagem de um arquivo PE.
- IATSusp: verifica se na tabela de endereços de arquivos importados existe referência para as seguintes APIs: OpensProcess, VirtualAllocEx, WriteProcessMemory, CreateRemoteThread, ReadProcessMemory, CreateProcess, WinExec, ShellExecute, HttpSendRequest, InternetReadFile, InternetConnect, CreateService e StartService.
- SuspiciousString: identifica se no código desmontado existe uma *string* suspeita. Uma *string* é considerada suspeita se ela constar no índice de *strings* suspeitas que definimos de acordo com (SIKORSKI, 2012) (HEXACORN, 2014). Este dicionário é constituído pelas seguintes *strings*: UrlDownloadToFile, GetTempPath, GetWindowsDirectory, GetSystemDirectory, WinExec, ShellExecute, IsBadReadPtr, IsBadWritePtr, CreateFile, DeleteFile, CreateHandle, CreateDirectory, CreateWindow, ReadFile, WriteFile, SetFilePointer, VirtualAlloc, GetProcAddress, LoadLibrary, Up, NumLock, Down, Right, Up, Left, PageUp, PageDown, GetLayout, SetLayout, gethostbyname, FindResource, LoadResource, WriteProcessMemory, GetModuleHandle, mscoree.dll, dddd, MMMM dd, yyyy, USER32.DLL, urlmon.dll, SeTakeOwnershipPrivilege, inflate, deflate, SeRestorePrivilege, SeTakeOwnershipPrivilege, SeRestorePrivilege, SeBackupPrivilege, RegCreateKeyExW, RegSaveKeyW, RegRestoreKeyW, RegOpenKeyExW, RegFlushKey, RegCloseKey, RegSetValueExW, RegDeleteValueW, RegQueryValueExW, RegDeleteKeyW, Process32FirstW, ReadProcessMemory, Process32NextW, WriteProcessMemory, VirtualAllocEx, CreateRemoteThread, VirtualFreeEx. Quando é encontrada uma *string* suspeita é atribuído o valor 1 para o atributo e quando não for encontrada é atribuído 0.
- FileSize: tamanho do arquivo.
- CRCSusp: caso o CRC calculado e o real sejam divergentes o arquivo é considerado como suspeito. Quando é considerado suspeito é atribuído o valor 1 e quando o arquivo não é considerado suspeito é atribuído o valor 0 para o atributo.

Para a análise dinâmica foram escolhidas as APIs do Windows que constituem as características do vetor de dados da análise dinâmica. Utilizamos para a escolha destes atributos o seguinte conjunto catalogado por (SIKORSKI, 2012):

- CreateFile: cria um arquivo ou abre um arquivo existente. Pode ser utilizado por um malware para despejar um arquivo no sistema.
- CreateMutex: cria uma exclusão mútua de objeto que pode ser usada por um *malware*.
- CreateProcess: cria e inicia um novo processo.
- CreateRemoteThread: utilizado para iniciar uma *thread* em um processo remoto. Um *malware* pode utilizá-lo para injetar código em um processo existente.
- CreateService: cria um serviço que pode ser iniciado em tempo de *boot*. Um *malware* pode utilizá-lo para persistência ou para carregar *drivers*.
- OpenFile: abre um arquivo.
- DeleteFile: exclui um arquivo.
- FindWindow: busca por uma janela aberta no *Desktop*.
- OpenMutex: abre um *handle* para um objeto com exclusão mútua. Um *malware* pode utilizar uma exclusão mútua para evitar a infecção de um sistema por diferentes instâncias de um mesmo *malware*. Exemplo: quando um *trojan* infecta um ambiente o primeiro passo é obter um *handle* para um mutex nomeado, se o processo falhar o processo do *malware* é encerrado.
- OpenSCManager: abre um *handle* para o gerenciador de controle de serviços. Isto permitirá ao *malware* interagir com os processos dos serviços do Windows, iniciando-os ou parando-os.
- ReadFile: lê um arquivo.
- ReadProcessMemory: usado para ler a memória de um processo remoto. Regiões válidas de memória podem ser copiadas utilizando esta API. Um *malware* pode fazer uso desta API para obter, por exemplo, o número do cartão de crédito de um usuário, utilizando um código de busca.

- RegDeleteKey: apaga uma chave de registro e subchaves.
- RegEnumKeyEx: enumera as subchaves de um registro aberto. Pode ser utilizado em conjunto com RegDeleteKey para apagar recursivamente chaves de registro.
- RegEnumValue: enumera os valores para a chave de registro aberta.
- CreateSection: cria um objeto de seção.
- RegOpenKey: abre um *handle* para controlar a leitura e edição de um registro.
- ShellExecute: utilizado para executar um outro programa.
- TerminateProcess: termina um processo e todas as suas *threads*.
- URLDownloadToFile: faz um *download* de *n bits* da Internet e os salva em um arquivo.
- WriteFile: grava os dados no dispositivo de saída.
- WriteProcessMemory: grava dados em um processo remoto. Utilizado na injeção de código.
- ZwMapViewOfSection: mapeia a visão de uma seção em um espaço de endereçamento virtual.
- LoadDll: função de baixo nível que carrega uma DLL em um processo. Pode indicar que o programa atua de forma furtiva.
- GetProcedureAddress: recupera o endereço de uma função carregada na memória. Usada para importar funções de outras DLLs em adição às importadas no cabeçalho do arquivo PE.
- OpenKey: abre um *handle* para controlar a leitura e edição de um registro.
- QueryValueKey: acessa os valores de uma chave de registro.
- IsDebuggerPresent: verifica se o processo está sendo depurado. Utilizado na detecção de *debuggers*.
- GetSystemMetrics: obtém informações de configurações do sistema.

- SetInformationFile: altera as informações de um arquivo.
- CreateMutant: um objeto mutante é criado e é aberto um *handle* para ele.
- OpenSection: abre um *handle* para uma seção.
- SetWindowsHookExA: define uma *hook function* que será chamada quando um evento ocorrer. Normalmente é utilizada por *keyloggers* e *spywares*.
- RegQueryValueEx: retorna o tipo e o valor para o nome específico associado a uma chave de registro.
- RegCloseKey: fecha o *handle* de uma chave de registro.
- OpenMutant: abre um *handle* para o objeto para execução exclusiva da instância do *malware*.
- LdrGetDllHandle: obtém o *handle* de um objeto.
- FreeVirtualMemory: libera uma região de páginas dentro do espaço de endereços virtuais de memória de um processo.

Os valores de cada um destes atributos é o número de vezes que cada uma das chamadas de API foram feitas pelo *malware* durante sua execução.

Além destes atributos que representam as chamadas de sistema executadas pelo *malware*, também incluímos os atributos DroppedFiles que é a contagem do número de arquivos que foram colocados no sistema, seja por meio de download ou de criação e inserção pelo *malware*, e nrProc que é a contagem do número total de processos executados pelo *malware*.

5.4 EXECUÇÃO DAS ANÁLISES ESTÁTICA E DINÂMICA DE MALWARES

Na etapa 4 foram executadas as análises estática e dinâmica dos *malwares*. Para iniciar a análise foram executados *scripts* de submissão dos arquivos de *malwares* e não *malwares* para a análise do pescanner e do Cuckoo Sandbox.

Para a análise estática foi executado um *script* que submete os arquivos das amostras de *malwares* e não *malwares* ao programa pescanner.py. Também, foi utilizado o aplicativo pyew para desmontar o código e obter *strings* no código.

Ao final das análises foram gerados dois arquivos com os resultados das análises estática e dinâmica respectivamente de todos os exemplos.

A figura 5.6 apresenta um extrato da análise estática de um arquivo.

```
Meta-data
================================================================================
File:    /home/reinaldo/binariesmw/VirusShare_e06535cea7363104a6cf095fe62987a2
Size:    57892 bytes
Type:    PE32 executable (GUI) Intel 80386, for MS Windows, UPX compressed
MD5:     e06535cea7363104a6cf095fe62987a2
SHA1:    57e8888e18d5f7a21e47d8f77b3c9e81e3e20e3d
ssdeep:  1536:y1Xqmqo/QmlR7AG+QRHxBTPfASFm7Inouy8oEY5qXX+:T5oQm/tXlASc7goutoLIXX+
Date:    0x49B9EFA6 [Fri Mar 13 05:31:18 2009 UTC]
EP:      0x428230 UPX1 1/3 [SUSPICIOUS]
CRC:     Claimed: 0x0, Actual: 0x12a3b [SUSPICIOUS]

Signature scans
================================================================================


Resource entries
================================================================================
Name                RVA      Size     Lang         Sublang                   Type
--------------------------------------------------------------------------------
RT_BITMAP           0x25264  0x1b2    LANG_ENGLISH SUBLANG_ENGLISH_US        data
RT_BITMAP           0x25418  0x190    *unknown*    SUBLANG_NEUTRAL           data
RT_ICON             0x29268  0x10a8   LANG_ENGLISH SUBLANG_ENGLISH_US        data
RT_ICON             0x2a314  0x468    LANG_ENGLISH SUBLANG_ENGLISH_US        GLS_BINARY_LSB_FIRST
RT_RCDATA           0x26ab8  0x10     LANG_NEUTRAL SUBLANG_NEUTRAL           data
RT_RCDATA           0x26ac8  0x118    LANG_NEUTRAL SUBLANG_NEUTRAL           data
RT_GROUP_ICON       0x2a780  0x22     LANG_ENGLISH SUBLANG_ENGLISH_US        MS Windows icon resource - 2 icons, 32x32, 256-colors
RT_MANIFEST         0x2a7a8  0x2e5    LANG_ENGLISH SUBLANG_ENGLISH_US        XML document text

Sections
================================================================================
Name        VirtAddr     VirtSize     RawSize      Entropy
--------------------------------------------------------------------------------
UPX0        0x1000       0x1c000      0x0          0.000000    [SUSPICIOUS]
UPX1        0x1d000      0xc000       0xc000       7.956590    [SUSPICIOUS]
.rsrc       0x29000      0x2000       0x1e00       5.033817
```

FIG. 5.6: Relatório de análise estática

Meta-data é a primeira seção da análise. Ela apresenta o nome do arquivo, conforme armazenado na base de dados, o tamanho do arquivo, o tipo do arquivo, os hashes MD5, SHA1 e ssdeep, a data de compilação do arquivo e se é uma data suspeita ou não, qual é o endereço do ponto de entrada do arquivo, em que seção está este endereço e se este endereço é suspeito, o CRC do arquivo, se seu cálculo é diferente do real e, se for, sua identificação como suspeito.

Scans Signatures é uma seção que apresenta detecções de assinaturas na base do ClamAv antivírus. Neste caso, nada foi encontrado.

Resource Entries contém informações sobre recursos que são utilizados como, por exemplo, os caracteres específicos de uma linguagem, chinês, ocidental, latino, figuras ou ícones utilizados, etc.

Sections contém as seções do arquivo e informações como o nome da seção, o endereço virtual, o tamanho do código em memória, o tamanho do código em disco e a entropia. Neste caso o arquivo é suspeito porque possui entropia inferior a 1 e maior que 7 e, também, porque o campo rawsize possui tamanho igual a 0. A entropia indica que o

arquivo foi empacotado com o aplicativo UPX. O empacotador é utilizado para ofuscar o código. O arquivo analisado possui três seções e não foi detectado nenhum endereço de importação suspeito na tabela IAT.

A figura 5.7 apresenta um extrato da análise estática mostrando as *strings* que foram recuperadas do arquivo.

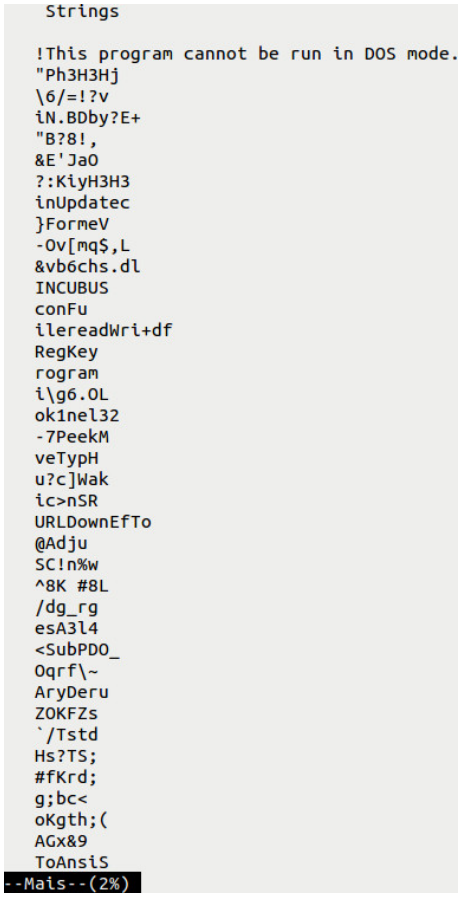
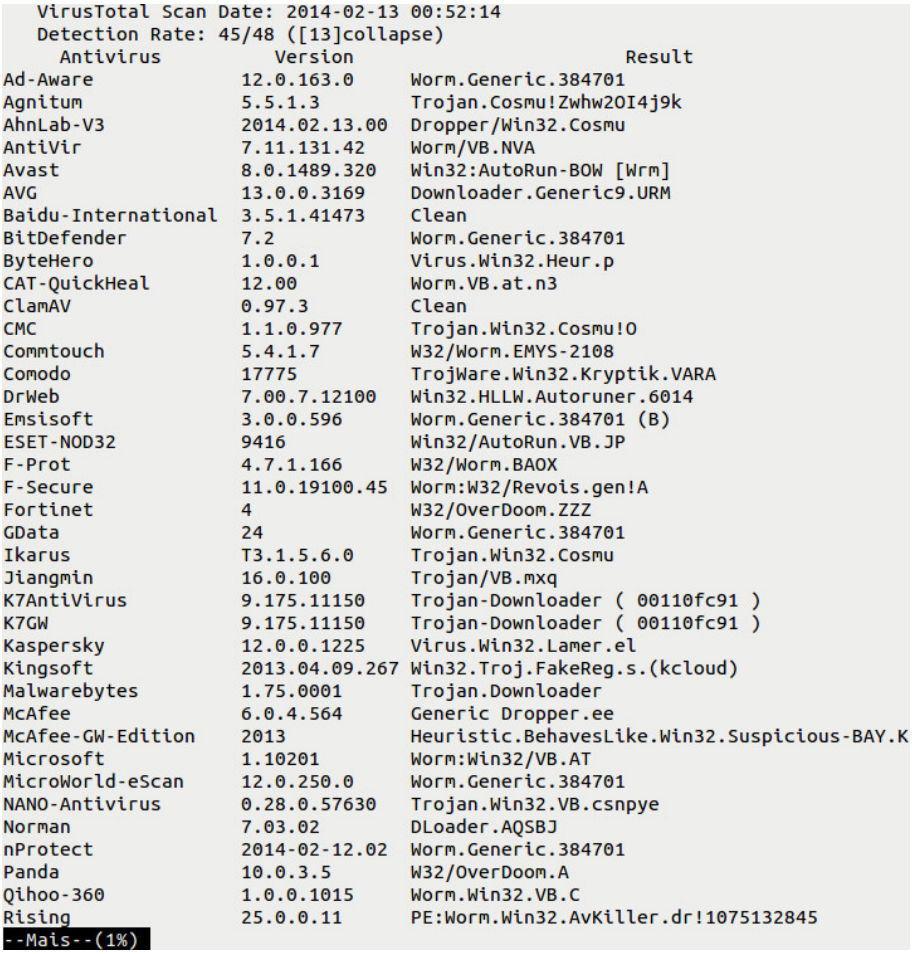


FIG. 5.7: Relatório de análise estática mostrando a recuperação de *strings*

Esta análise gerou, neste caso, uma grande relação de *strings* que são muito úteis para o trabalho do analista de *malwares*. Em nosso trabalho, utilizamos um dicionário que busca encontrar *strings* que podem ser consideradas como suspeitas. Esta lista pode ser ampliada, conforme a experiência for sendo adquirida no processo de análise.

A figura 5.8 mostra um extrato da análise estática mostrando a taxa de detecção e a nomeação do arquivo por alguns antivírus.

Podemos verificar, que o arquivo que utilizamos como exemplo, foi detectado como *malware* por 45 *softwares* antivírus. O total de antivírus que o analisaram foi 48 e, portanto, três não foram capazes de detectá-lo. Verificamos também, que cada um dos



```
   VirusTotal Scan Date: 2014-02-13 00:52:14
   Detection Rate: 45/48 ([13]collapse)
     Antivirus          Version                        Result
Ad-Aware              12.0.163.0      Worm.Generic.384701
Agnitum               5.5.1.3         Trojan.Cosmu!Zwhw2OI4j9k
AhnLab-V3             2014.02.13.00   Dropper/Win32.Cosmu
AntiVir               7.11.131.42     Worm/VB.NVA
Avast                 8.0.1489.320    Win32:AutoRun-BOW [Wrm]
AVG                   13.0.0.3169     Downloader.Generic9.URM
Baidu-International   3.5.1.41473     Clean
BitDefender           7.2             Worm.Generic.384701
ByteHero              1.0.0.1         Virus.Win32.Heur.p
CAT-QuickHeal         12.00           Worm.VB.at.n3
ClamAV                0.97.3          Clean
CMC                   1.1.0.977       Trojan.Win32.Cosmu!O
Commtouch             5.4.1.7         W32/Worm.EMYS-2108
Comodo                17775           TrojWare.Win32.Kryptik.VARA
DrWeb                 7.00.7.12100    Win32.HLLW.Autoruner.6014
Emsisoft              3.0.0.596       Worm.Generic.384701 (B)
ESET-NOD32            9416            Win32/AutoRun.VB.JP
F-Prot                4.7.1.166       W32/Worm.BAOX
F-Secure              11.0.19100.45   Worm:W32/Revois.gen!A
Fortinet              4               W32/OverDoom.ZZZ
GData                 24              Worm.Generic.384701
Ikarus                T3.1.5.6.0      Trojan.Win32.Cosmu
Jiangmin              16.0.100        Trojan/VB.mxq
K7AntiVirus           9.175.11150     Trojan-Downloader ( 00110fc91 )
K7GW                  9.175.11150     Trojan-Downloader ( 00110fc91 )
Kaspersky             12.0.0.1225     Virus.Win32.Lamer.el
Kingsoft              2013.04.09.267  Win32.Troj.FakeReg.s.(kcloud)
Malwarebytes          1.75.0001       Trojan.Downloader
McAfee                6.0.4.564       Generic Dropper.ee
McAfee-GW-Edition     2013            Heuristic.BehavesLike.Win32.Suspicious-BAY.K
Microsoft             1.10201         Worm:Win32/VB.AT
MicroWorld-eScan      12.0.250.0      Worm.Generic.384701
NANO-Antivirus        0.28.0.57630    Trojan.Win32.VB.csnpye
Norman                7.03.02         DLoader.AQSBJ
nProtect              2014-02-12.02   Worm.Generic.384701
Panda                 10.0.3.5        W32/OverDoom.A
Qihoo-360             1.0.0.1015      Worm.Win32.VB.C
Rising                25.0.0.11       PE:Worm.Win32.AvKiller.dr!1075132845
--Mais--(1%)
```

FIG. 5.8: Relatório de detecção e nomeação de antivírus

antivírus, nomeiam o *malware* de formas diferentes.

A figura 5.9 apresenta um extrato da análise dinâmica do arquivo acima citado.

No extrato é possível verificar o nome do arquivo que foi analisado o seu número de processo e do processo pai, as chamadas de sistema e funções que foram executadas.

5.5 PREPARAÇÃO DOS DADOS

Os arquivos gerados na etapa 5 contêm as informações tratadas e consolidadas e que servem de entrada para o processo de classificação. Para a seleção dos dados utilizamos um programa em linguagem C, preparados.c, que lê os arquivos de entrada, referentes às análises estática e dinâmica e seleciona os atributos de interesse para a análise, conforme definido na etapa 3. Assim, são gerados dois conjuntos de dados. Um referente às informações coletadas pela análise estática e outro com as informações coletadas a partir da

FIG. 5.9: Relatório de análise dinâmica

análise dinâmica. Estes dados são filtrados de tal forma a permitir a unificação e posterior submissão ao algoritmo de aprendizado de máquina.

A figura 5.10 mostra um extrato do arquivo com extensão .dat que possui os valores dos atributos resultantes da seleção.

5.6 PREPARAÇÃO DOS ALGORITMOS DE APRENDIZADO DE MÁQUINA

Na etapa 6 foram instalados e configurados os aplicativos e algoritmos de aprendizado de máquina, para a classificação e predição dos exemplos. Foi utilizado nesta fase o *framework* de aprendizado de máquina FAMA. Os arquivos criados na fase 5, com os valores dos

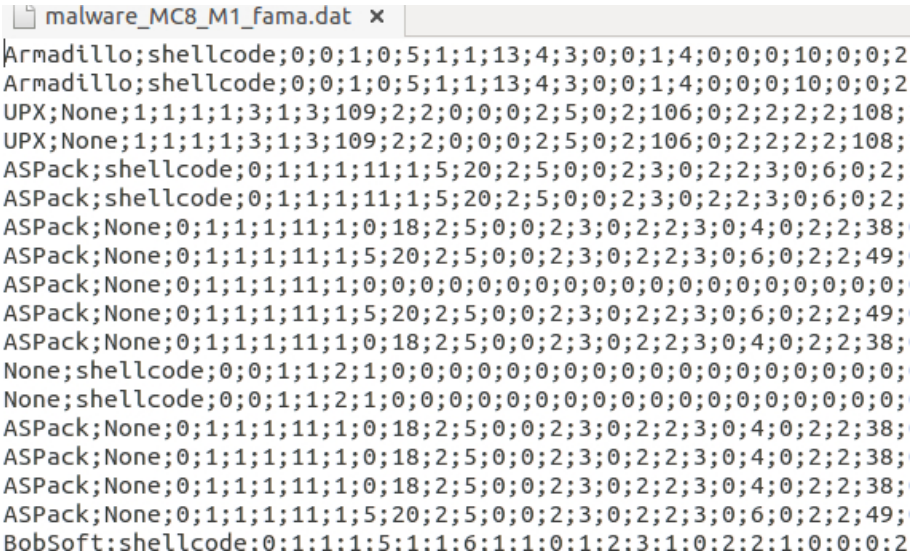


FIG. 5.10: Exemplo do arquivo .dat

atributos, foram convertidos para o formato csv. Estes arquivos contêm os dados que são inseridos no vetor de entrada de dados do algoritmo de aprendizado de máquina. Os algoritmos de aprendizado de máquina escolhidos para a tarefa de classificação foram o C5.0 e o Random Forest. O Fator de Confiança utilizado no algoritmo C5.0 foi igual a 35% ou 0.35 e o número de árvores e de atributos utilizados pelo Random Forest foram 100 e 6, respectivamente. Para escolher estes valores, nos primeiros experimentos (I até VI), executamos repetidamente os algoritmos utilizando diversas configurações e, após encontrar as definições que produziam melhor desempenho, definimos estes valores como padrões.

5.7 ANÁLISE DOS RESULTADOS

Na última etapa os dados obtidos na fase de aprendizado de máquina foram organizados e comparados. Os resultados foram obtidos utilizando o método de validação cruzada com 10 partições. As métricas que utilizamos na apuração dos resultados foram a acurácia, a precisão, a revocação, F-measure e concordância Kappa.

6 RESULTADOS

A finalidade deste trabalho é verificar se a classificação de *malwares* que utiliza o método unificado possui melhor desempenho do que uma análise isolada. Foram realizados 9 experimentos. Os experimentos I, II e III comparam o desempenho da identificação de *malwares* utilizando a classificação binária. São considerados respectivamente, os atributos das abordagens dinâmica, estática e unificada. Os experimentos de IV até IX consideram o problema da classificação utilizando a categorização definida pelo CERT.BR para tipos de *malwares*: vírus, worm, bot, trojan, spyware, backdoor e rootkit. Além destas, também utilizamos a categoria benigno, pois um artefato pode ser um não *malware*. A diferença entre as duas séries de experimentos, IV-VI e VII-IX é o uso de diferentes atributos dos *malwares*. Na primeira série a classificação multiclasses a priori é utilizada na indução. Na segunda série, a classe predita nos experimentos I,II e III, *malware* ou não *malware* é também utilizada como entrada para o processo de classificação.

Foram escolhidos aleatoriamente 6292 exemplos a partir do conjunto de exemplos e suas quantidades e proporções estão descritas na tabela 6.1.

TAB. 6.1: Experimentos

EXP	ANÁLISE	CLASSE	A PRIORI	% APRIORI
I, II, III	Unificada, Estática, Dinâmica	MALIGNO	3633	57,74%
		BENIGNO	2659	42,26%
IV até IX	Unificada, Estática, Dinâmica	TROJAN	1941	30,85%
		VIRUS	585	9,30%
		WORM	747	11,87%
		BACKDOOR	111	1,76%
		ROOTKIT	97	1,54%
		BOT	47	0,75%
		SPYWARE	105	1,67%
		BENIGNO	2659	42,26%

Consideramos a análise de acordo com dois pontos de vista. Inicialmente, testamos as capacidades de classificação e de predição utilizando a classificação binária (Experimentos I, II e III) e posteriormente o processo de classificação multiclasses (Experimentos IV até IX).

Utilizamos dois algoritmos de aprendizado de máquina para classificar os exemplos: o

C5.0 e o Random Forest.

O número total de atributos utilizados na análise unificada foram 50, na análise estática 10 e na análise dinâmica 40. 1 atributo é o rótulo de classe. O fator de confiança (CF) utilizado para testar a efetividade da pós-poda do algoritmo C5.0 foi igual a 0.35. O número de árvores do Random Forest foi definido como 100 e o número de atributos aleatórios utilizados na construção das árvores do comitê foi definido como 6.

Todas estas configurações dos algoritmos de aprendizado de máquina foram definidas nos primeiros experimentos e depois fixados para todas as séries.

Todos os experimentos foram validados utilizando o método de validação cruzada com 10 partições e as métricas utilizadas para avaliar o desempenho das classificações foram a acurácia, a revocação, a precisão, a média harmônica entre a precisão e a revocação e a concordância Kappa.

A tabela 6.2 e figura 6.1 resumem os resultados obtidos pelas análises utilizando classes binárias e múltiplas. Os melhores resultados estão destacados em negrito.

TAB. 6.2: Resultados dos Experimentos - Acurácia e concordância Kappa

EXP	ANALISE	ACURACIA C5.0	ACURACIA Random Forest	KAPPA C5.0	KAPPA Random Forest
I	UNIFICADA	91,91%	**95,75%**	83,45%	91,30%
II	ESTÁTICA	86,82%	**89,91%**	72,70%	79,24%
III	DINÂMICA	88,67%	**93,55%**	76,50%	86,80%
IV	UNIFICADA	82,96%	**88,94%**	75,50%	84,14%
V	ESTÁTICA	75,45%	**79,37%**	64,44%	70,37%
VI	DINÂMICA	80,13%	**86,52%**	71,40%	80,66%
VII	UNIFICADA	85,49%	**93,02%**	79,19%	90,01%
VIII	ESTÁTICA	77,80%	**82,53%**	67,77%	74,94%
IX	DINÂMICA	86,29%	**91,32%**	80,71%	87,58%

Podemos observar que os índices medidos de acurácia demonstram que os melhores resultados correspondem àqueles testes que utilizaram a análise unificada. A acurácia da análise para o problema de classificação binária foi igual a 91.91%, utilizando o classificador C5.0 e a concordância Kappa foi igual a $83,45\%$. Por sua vez, os índices obtidos com o classificador Random Forest foram superiores aos obtidos pelo classificador C5.0. A acurácia medida para o Random Forest foi igual a $95,75\%$ e a concordância Kappa foi igual a $91,30\%$. Verificando a tabela 3.2 podemos inferir que a concordância obtida foi quase perfeita para os dois algoritmos de aprendizado de máquina. A análise estática, por outro lado, obteve os mais fracos resultados.

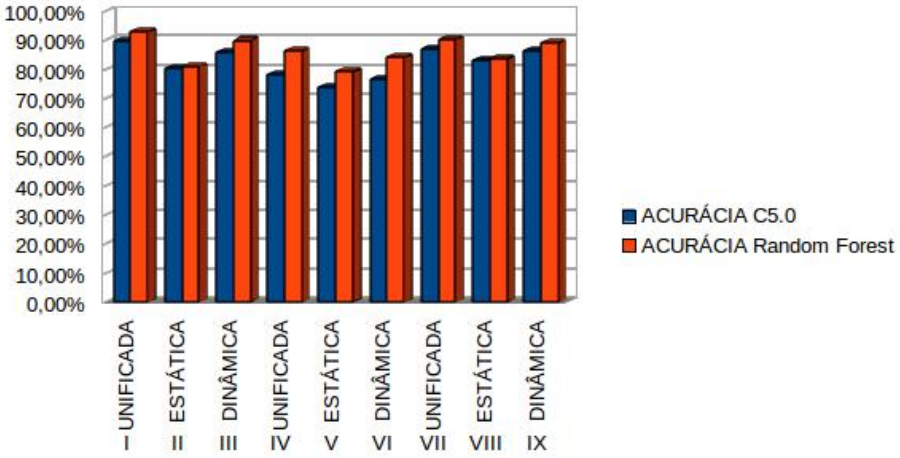


FIG. 6.1: Resultados (Acurácia)

Os gráfico da figura 6.1 mostra que em todos os experimentos os índices de acurácia do algoritmo Random Forest foram superiores aos obtidos pelo algoritmo C5.0.

Para o problema de multiclassificação, o experimento VII obteve a melhor acurácia, igual a 93.02%, utilizando o Random Forest, e acurácia igual a $85,49\%$ utilizando o C5.0. A concordância Kappa obteve índices iguais a $90,01\%$ para o Random Forest e $79,19\%$ para o C5.0. Neste caso, a concordância pode ser considerada quase perfeita para ambos os resultados obtidos. Também, pode-se observar que para estes experimentos o desempenho da análise unificada foi superior ao das análises isoladas.

Para melhor analisar os resultados dos experimentos IV até IX, utilizamos as métricas de F-measure, precisão e revocação para comparar os resultados das classes. Os melhores resultados foram obtidos pelo experimento VII e são apresentados na tabela 6.3.

TAB. 6.3: Resultados - Experimento VII

EXP	AN.	CLASS	PREC C5.0	PREC R Forest	REV C5.0	REV R Forest	F-MEAS C5.0	F-MEAS R Forest
VII	**UNIF**	BENIG	94,40%	**99,90%**	96,30%	**99,90%**	0,9534	**0,9990**
		TROJ	78,90%	**87,20%**	82,50%	**92,70%**	0,8066	**0,8987**
		VIRUS	70,60%	**85,80%**	60,30%	**76,20%**	0,6504	**0,8072**
		WORM	88,30%	**92,20%**	80,20%	**86,60%**	0,8406	**0,8931**
		BACKD	75,20%	**95,50%**	71,20%	**75,70%**	0,7315	**0,8446**
		ROOTK	71,90%	**82,00%**	84,50%	**93,80%**	0,7769	**0,8750**
		BOT	42,30%	**78,80%**	23,40%	**55,30%**	0,3013	**0,6499**
		SPYW	72,90%	**88,80%**	89,50%	**98,10%**	0,8035	**0,9322**

A figura 6.2 permite uma visão geral dos resultados obtidos para o experimento VII, comparando os resultados de revocação obtidos.

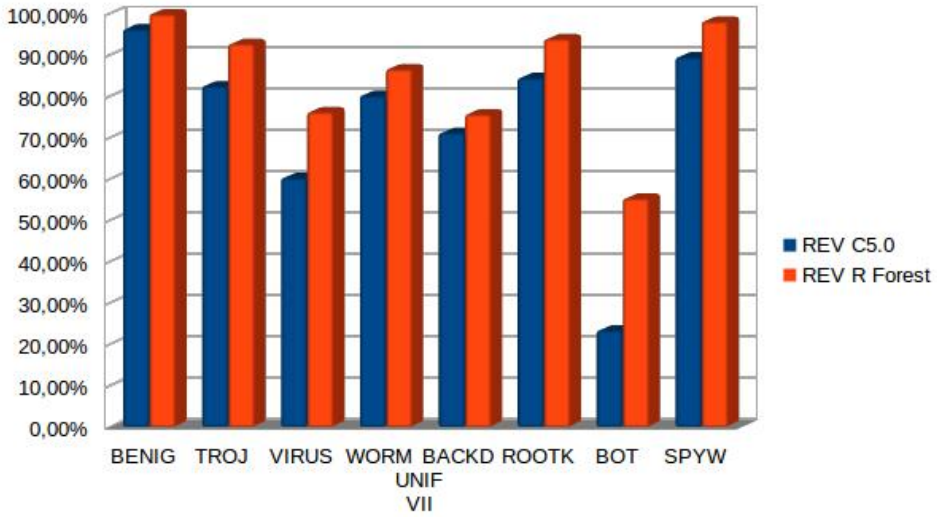


FIG. 6.2: Resultados (Revocação)

Os melhores resultados de *revocação* foram os obtidos para os tipos de classes de *malwares* trojan, worm, spyware e rootkit e, em geral, os resultados de revocação do Random Forest foram superiores.

As classes benigno, spyware, rootkit e trojan obtiveram revocação acima de 90%. Isto significa que de todos os exemplos classificados a priori como benignos, spywares, rootkits ou trojan, mais de 90% foram identificados pelo Random Forest.

A classe Bot, que é uma classe definida pelo CERT.BR (CERT.BR, 2013) e não por Szor (SZOR, 2005), obteve o menor desempenho na classificação. Podemos inferir que isto pode ser explicado pelo pequeno número de exemplos existentes na amostra.

7 CONCLUSÃO E SUGESTÕES DE TRABALHOS FUTUROS

A evolução das técnicas empregadas no desenvolvimento de *malwares*, em especial aquelas utilizadas com objetivos de impedir ou dificultar a identificação de códigos maliciosos em sistemas computacionais, criam obstáculos para o desenvolvimento de *softwares* antivírus. Técnicas de evasão têm sido empregadas com sucesso e, devido ao crescente volume de ataques e do interesse do uso de códigos maliciosos para atividades de espionagem, hacktivismo e obtenção ilícita de vantagens financeiras e informações, novos métodos para análise e detecção de *malwares* têm sido desenvolvidos. Estes métodos abordam o estudo das características e comportamento dos códigos maliciosos e técnicas utilizadas para automatizar a sua classificação. A classificação automática dos *malwares* pode ser realizada utilizando algoritmos de aprendizado de máquina. Um código malicioso pode ser classificado como um *malware* ou *não malware*, ou de forma mais complexa, em tipos de *malwares* ou em famílias de *malwares*. Estas classificações complexas são muito difíceis de serem realizadas devido a grande proliferação, capacidade de mutação e variabilidade dos *malwares*. Com o objetivo de colaborar com a tarefa de identificação dos *malwares* utilizamos neste trabalho um método de unificação das análises estática e dinâmica e classificação automática de *malwares* utilizando algoritmos de aprendizado de máquina.

Utilizamos uma base de dados de *malwares* obtida do *site* VirusShare e fizemos uma análise de cada um dos exemplos da base no site do VirusTotal, verificando e confirmando a classificação dos arquivos como *malwares*. Também classificamos estes arquivos em tipos de *malwares* de acordo com a classificação dada pela maioria dos antivírus da base do VirusTotal.

Selecionamos as características dos *malwares* obtendo informações das análises estática e dinâmica. Para a seleção das características utilizamos técnicas como o cálculo de entropia dos dados para identificação de possíveis arquivos ofuscados, *strings* suspeitas no código, chamadas de sistemas suspeitas, datas inválidas de criação do arquivo, *checksum* inválido de arquivo, endereços de seções suspeitos, frequências de chamadas de APIs pelos arquivos em execução.

Construímos vetores de dados das análises estática, dinâmica e unificada a partir das características calculadas e selecionadas a partir dos dados da análise acima citados.

Foram implementados no FAMA os algoritmos de aprendizado de máquina C5.0 e Random Forest.

Mostramos que a unificação das análises estática e dinâmica de *malwares* pode ser utilizada para diminuir a possibilidade de sucesso da evasão de *malwares*. Verificamos que em todos os cenários montados, a técnica de unificação das análises estática e dinâmica de *malwares* apresentou melhor acurácia que as técnicas isoladas de análise estática e dinâmica. Os índices de acurácia obtidos para a análise unificada foram superiores a 90%, em ambas as abordagens de classificação, utilizando a classificação binária e a classificação multiclasses. Estes índices estão de acordo com outras abordagens de análise propostas em outros trabalhos acadêmicos. Os métodos de classificação utilizados na classificação automática de *malwares* foram baseados em árvores de decisão, utilizando o algoritmo de aprendizado de máquina C5.0 e o método de comitê, utilizando o algoritmo Random Forest, implementados no *framework* de aprendizado FAMA. Os melhores resultados foram obtidos com o algoritmo Random Forest.

Além da acurácia, utilizamos a média harmônica de precisão e revocação para medir o desempenho dos algoritmos. Estes índices mostram que houve um aumento no desempenho quando utilizamos a técnica unificada.

Neste nosso trabalho utilizamos uma abordagem *batch* para a análise de *malwares*. Mas, sabemos que desenvolvedores de *malwares* aperfeiçoam diariamente suas técnicas para impedir que os especialistas e sistemas de segurança detectem suas ações. Buscam sobretudo explorar a capacidade de interligação dos sistemas via redes para disseminar e ampliar seus ataques. Uma abordagem *online*, que seja capaz de aprender um novo exemplo, de forma incremental, poderia ser útil para mitigar a complexidade computacional dos métodos *batch* e realizar a predição *on line* de códigos maliciosos. Outras sugestões de trabalhos são a aplicação de nossa metodologia utilizando outros algoritmos de aprendizado de máquina como o AdaBoost e o SVM.

Também seria interessante a aplicação da abordagem unificada em outras plataformas que utilizem outros sistemas operacionais, como, por exemplo, Android, utilizar outras bases de malwares e aplicar o método em outros formatos de arquivos como pdf e documentos do Office.

A seleção de características pode ser melhorada e um método de seleção baseado no reconhecimento de padrões de sequências de símbolos no código desmontado pode ser útil na definição de novas características para classificação de *malwares*.

A integração do processo de análise e de classificação em um único *framework* também pode ser uma solução para melhorar a classificação de *malwares*.

8 REFERÊNCIAS BIBLIOGRÁFICAS

BRANCO, R. R., BARBOSA, G. N. e NETO, P. D. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. Em *Black Hat Technical Security Conf.(Las Vegas, Nevada*, 2012.

BREIMAN, L. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

BREIMAN, L. Random forests. *Machine learning*, 45(1):5–32, 2001.

BUJLOW, T., RIAZ, T. e PEDERSEN, J. M. A method for classification of network traffic based on c5. 0 machine learning algorithm. Em *Computing, Networking and Communications (ICNC), 2012 International Conference on*, págs. 237–241. IEEE, 2012.

CAFFÉ, M. I. R., PEREZ, P. S. e BARANAUSKAS, J. A. Avaliação do algoritmo de stacking em dados biomédicos.

CERIANI, L. e VERME, P. The origins of the gini index: extracts from variabilità e mutabilità (1912) by corrado gini. *The Journal of Economic Inequality*, 10(3):421–443, 2012.

CERT.BR. *Cartilha de Segurança para Internet.* Centro de Estudos, Resposta e Tratamento de Incidentes de Segurança no Brasil, Brasil, dezembro 2013. URL `http://cartilha.cert.br/malware`.

CHEN, T. M. e ROBERT, J.-M. The evolution of viruses and worms. *Statistical Methods in Computer*, 2004.

CHRISTODORESCU, M. e JHA, S. Static analysis of executables to detect malicious patterns. Technical report, DTIC Document, 2006.

COHEN, F. B. e COHEN, D. F. *A short course on computer viruses.* John Wiley & Sons, Inc., 1994.

COHEN, J. A coefficient of agreement for nominal scales. educational and psychological measurement. *Educational and Psychological Measurement*, 20:37–46, 1960.

CORTES, C. e VAPNIK, V. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

COZZOLINO, M. F. Detecção de variantes metamórficas de malware por comparação de códigos normalizados. 2012.

CTIR. Relatório de estatísticas de incidentes de rede na administração pública federal - 2 trimestre 2014. Technical report, Centro de Tratamento de Incidentes de Segurança de Redes de Computadores da Administração Pública Federal, 2014.

DE ANDRADE, C. A. B., MELLO, C. G. e DUARTE, J. C. Malware automatic analysis. *Computational Intelligence and 11th Brazilian Congress on Computational Intelligence (BRICS-CCI & CBIC)*, págs. 681–686, 2013.

DE ARAÚJO JORGE, B. W. G. Estados unidos, poder cibernético e a" guerra cibernética": Do worm stuxnet ao malware flame/skywiper–e além. *Meridiano 47-Boletim de Análise de Conjuntura em Relações Internacionais*, 13(131), 2012.

DUARTE, J. C. O algoritmo boosting at start e suas aplicaçoes. 2009.

EAGLE, C. *The IDA pro book: the unofficial guide to the world's most popular disassembler*. No Starch Press, 2008.

ELKAN, C. Boosting and naive bayesian learning. Technical report, Technical Report CS97-557, University of California, San Diego, 1997.

FAMA. Framework de aprendizado de máquina, 2014. URL `https://code.google.com/p/fama/`.

FERRIE, P. e SZÖR, P. Hunting for metamorphic. *Virus*, págs. 123–143, 2001.

FORTINET. Understanding how file size affects malware detection, 2014. URL `http://www.fortinet.com/sites/default/files/whitepapers/MalwareFileSize.pdf`.

GASPAR, P. Pragas eletrônicas: ainda não estamos livres delas. 2007.

GOYAL, R., SHARMA, S., BEVINAKOPPA, S. e WATTERS, P. Obfuscation of stuxnet and flame malware. *Latest Trends in Applied Informatics and Computing*, 2012.

HAN, J. e KAMBER, M. *Data Mining. Concepts and Techniques*. Morgan Kauffman, 30 Corporate Drive, Suite 400, Burlington, MA 01803, USA, 2 edition, 2006.

HARVEI, P. Exiftool by phil harvey. read, write and edit meta information!, 2014. URL `http://www.sno.phy.queensu.ca/ phil/exiftool/`.

HEXACORN. Hexdive 0.4, 2014. URL `http://www.hexacorn.com/blog/2012/08/`.

H.MALIN, C., CASEY, E., AQUILINA, J. M. e ROSE, C. W. *Malware Forensics Field Guide For Windows Systems*. Elsevier, 225 Wyman Street, Waltham, MA 02451, USA, 2012.

HO, T. K. Random decision forests. Em *Document Analysis and Recognition, 1995., Proceedings of the Third International Conference on*, volume 1, págs. 278–282. IEEE, 1995.

HUNT, E. B., MARIN, J. e STONE, P. J. *Experiments in induction*. Academic Press, New York, USA, 1966.

KOLTER, J. Z. e MALOOF, M. A. Learning to detect malicious executables. *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2004.

KONG, D. e YAN, G. Discriminant malware distance learning on structural information for automated malware classification. *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, págs. 1357–1365, 2013.

KONSTANTINOU, E. e WOLTHUSEN, S. Metamorphic virus: Analysis and detection. *Royal Holloway University of London*, 15, 2008.

LABDCIBER-IME. Laboratório de defesa cibernética do ime, 2014. URL `http://defesacibernetica.ime.eb.br/`.

LANDIS, J. R. e KOCH, G. G. The measurement of observer agreement for categorical data. *biometrics*, 33(1):159–174, 1977.

LIGH, M., ADAIR, S., HARTSTEIN, B. e RICHARD, M. *Malware Analyst's Cookbook and DVD: Tools and Techniques for Fighting Malicious Code*. Wiley Publishing, 2010.

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*, págs. 290–299. ACM, 2003.

LORENA, A. C. e DE CARVALHO, A. C. Uma introdução às support vector machines. *Revista de Informática Teórica e Aplicada*, 14(2):43–67, 2007.

LYDA, R. e HAMROCK, J. Using entropy analysis to find encrypted and packed malware. *IEEE Security & Privacy*, 5(2):40–45, 2007.

MITCHELL, T. M. *Machine Learning*. McGraw-Hill Science/Engineering, Math, Redmond,WA, 1997.

MOHAISEN, A., WEST, A. G., MANKIN, A. e ALRAWI, O. Chatter: Exploring classification of malware based on the order of events. 2014.

MONARD, M. C. e BARANAUSKAS, J. A. *Conceitos sobre Aprendizado de Máquina*, chapter Conceitos sobre Aprendizado de Máquina, pág. 90. Editora Manole, Ltda, Barueri, SP, 2003.

NEUMANN, J. V. e BURKS, A. W. Theory of self-reproducing automata. 1966.

OBERHEID, J. C. Virustotal python submission script, 2014. URL `https://jon.oberheide.org/blog/2008/11/20/virustotal-python-submission-script/`.

ODP. Dmoz - open directory project - data formats, 2014. URL `http://www.dmoz.org/Computers/Data_Formats/`.

OKTAVIANTO, D. e MUHARDIANTO, I. *Cuckoo Malware Analysis*. Packt Publishing Ltd, 2013.

QUINLAN, J. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

QUINLAN, J. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

QUINLAN, R. See5/c5.0. rulequest research: datamining tools, 2010. URL http://www.rulequest.com.

RAMAN, K. Selecting features to classify malware. *InfoSec Southwest 2012*, 2012.

ROEA, B. P., YANGA, H. e ZHUB, J. Boosted decision trees, a powerful event classifier. *trees*, 2:2–2, 2005.

RUSSELL, S., NORVIG, P. e INTELLIGENCE, A. A modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, 25, 1995.

SAMI, A., YADEGARI, B., RAHIMI, H., PEIRAVIAN, N., HASHEMI, S. e HAMZE, A. Malware detection based on mining api calls. Em *Proceedings of the 2010 ACM Symposium on Applied Computing*, págs. 1020–1025. ACM, 2010.

SCHAPIRE, R. E. e FREUND, Y. *Boosting: Foundations and algorithms*. MIT Press, 2012.

SECURITY, P. Malware básico - a evolução do malware da sua origem em 1949, aos dias de hoje, 2014. URL http://www.pandasecurity.com/brazil/homeusers/security-info/classic-malware/.

SHANNON, C. E. Matematical theory of communication. *Bell System Techinical Journal*, 27(379-423), 1948.

SIDDIQUI, M., WANG, M. C. e LEE, J. Detecting internet worms using data mining techniques. *Journal of Systemics, Cybernetics and Informatics*, 6(6), 2008.

SIKORSKI, M. e HONIG, A. *Pratical Malware Analysis*. William Pollock, San Francisco, CA, 2012.

SISTEMAS, H. Virus total. *URL: http://www.virustotal.com*, 2014.

STEVANOVIC, M. Linux toolbox. Em *Advanced C and C++ Compiling*, págs. 243–276. Springer, 2014.

SZOR, P. *The art of computer virus research and defense*. Pearson Education, 2005.

UFF. Padrões, normas, protocolos e formatos de arquivos digitais, 2014. URL http://www.professores.uff.br/marcondes/antigo/TIs-PADROES-DIGITALIZACAO.pdf.

UGARTE-PEDRERO, X., SANTOS, I., SANZ, B., LAORDEN, C. e BRINGAS, P. G. Countering entropy measure attacks on packed software detection. Em *Consumer Communications and Networking Conference (CCNC), 2012 IEEE*, págs. 164–168. IEEE, 2012.

VON ZUBEN, I.-P. F. J. e ATTUX, R. R. Árvores de decisão.

WEKA. Weka 3: Data mining software in java - university of waikato, 2014. URL http://www.cs.waikato.ac.nz/ml/weka/.

WILLEMS, C., HOLZ, T. e FREELING, F. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security and Privacy*, 5(2):32–39, 2007.

WITTEN, I. H., FRANK, E. e HALL, M. A. *Data Mining. Practical Machine Learning Tools and Techniques*. Morgan Kauffman, 30 Corporate Drive, Suite 400, Burlington, MA 01803, USA, 3 edition, 2011.

YOU, I. e YIM, K. Malware obfuscation techniques: A brief survey. Em *BWCCA*, págs. 297–300, 2010.

ZABIDI, M. N. A., MAAROF, M. A. e ZAINAL, A. Malware analysis with multiple features. Em *Computer Modelling and Simulation (UKSim), 2012 UKSim 14th International Conference on*, págs. 231–235. IEEE, 2012.