UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

DETECÇÃO DE VARIANTES METAMÓRFICAS DE MALWARE POR COMPARAÇÃO DE CÓDIGOS NORMALIZADOS

MARCELO FREIRE COZZOLINO

ORIENTADOR: EDUARDO JAMES PEREIRA SOUTO
CO-ORIENTADOR: FLÁVIO ELIAS GOMES DE DEUS

DISSERTAÇÃO DE MESTRADO EM ENGENHARIA ELÉTRICA

PUBLICAÇÃO: 098/12 ENE/PG

BRASÍLIA / DF: FEVEREIRO/2012

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

DETECÇÃO DE VARIANTES METAMÓRFICAS DE MALWARE POR COMPARAÇÃO DE CÓDIGOS NORMALIZADOS

MARCELO FREIRE COZZOLINO

DISSERTAÇÃO DE MESTRADO SUBMETIDA AO DEPARTAMENTO DE ENGENHARIA ELÉTRICA DA FACULDADE DE TECNOLOGIA DA UNIVERSIDADE DE BRASÍLIA, COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE PROFISSIONAL EM ENGENHARIA ELÉTRICA, ÁREA DE CONCENTRAÇÃO EM INFORMÁTICA FORENSE E SEGURANÇA DA INFORMAÇÃO.

APROVADA POR:

EDUARDO JAMES PEREIRA SOUTO, (ICOMP/ UFAM)
(ORIENTADOR)

PROF. ANDERSON CLAYTON ALVES NASCIMENTO (ENE/ UNB)
(EXAMINADOR INTERNO)

PROF. DJAMEL FAWZI HADJ SADOK (CIN / UFPE)
(EXAMINADOR EXTERNO)

DATA: BRASÍLIA/DF, 27 DE FEVEREIRO DE 2012.

FICHA CATALOGRÁFICA

COZZOLINO, MARCELO FREIRE
Detecção de Variantes Metamórficas de Malware por Comparação de Códigos Normalizados
Distrito Federal 2012.
xiv, 38p., 297 mm (ENE/FT/UnB, Mestre, Engenharia Elétrica, 2012).

Dissertação de Mestrado – Universidade de Brasília, Faculdade de Tecnologia. Departamento de Engenharia Elétrica.

1. Malware 2. Tokens
3. Códigos Normalizados 4.Metamorfismo
5. Ofuscação

I. ENE/FT/UnB II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

COZZOLINO, M.F. (2012). Detecção de Variantes Metamórficas de Malware por Comparação de Códigos Normalizados. Dissertação de Mestrado, Publicação ENE/PG - 098/12, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 52p.

CESSÃO DE DIREITOS

NOME DO AUTOR: Marcelo Freire Cozzolino
TÍTULO DA DISSERTAÇÃO: Detecção de Variantes Metamórficas de Malware por Comparação de Códigos Normalizados.
GRAU/ANO: Mestre/2012.

É concedida à Universidade de Brasília permissão para reproduzir cópias desta Dissertação de Mestrado e para emprestar ou vender tais cópias somente para propósitos acadêmicos e científicos. Do mesmo modo, a Universidade de Brasília tem permissão para divulgar este documento em biblioteca virtual, em formato que permita o acesso via redes de comunicação e a reprodução de cópias, desde que protegida a integridade do conteúdo dessas cópias e proibido o acesso a partes isoladas desse conteúdo. O autor reserva outros direitos de publicação e nenhuma parte deste documento pode ser reproduzida sem a autorização por escrito do autor.

____

Marcelo Freire Cozzolino

A meu pai, Luiz Tinoco Cozzolino

AGRADECIMENTOS

Agradeço ao meu orientador, Prof. Dr. Eduardo James Pereira Souto, ao colaborador Prof. Gilbert Martins, ao meu co-orientador prof. Dr. Flavio Elias Gomes de Deus, ao Departamento de Polícia Federal, ao Programa Nacional de Segurança Pública com Cidadania (PRONASCI) e ao povo brasileiro, sem os quais este trabalho não teria sido possível.

RESUMO

DETECÇÃO DE VARIANTES METAMÓRFICAS DE MALWARE POR COMPARAÇÃO DE CÓDIGOS NORMALIZADOS

Autor: Marcelo Freire Cozzolino

Orientador: Eduardo James Pereira Souto

Co-orientador: Flavio Elias Gomes de Deus

Programa de Mestrado em Engenharia Elétrica

Brasília, Fevereiro de 2012

Malware é um termo usado para descrever todos os tipos de *software* maliciosos como vírus, *worms ou trojan horses*. Para combater tais ameaças, muitas técnicas e ferramentas têm sido empregadas como os anti-* (*anti-virus, anti-malware, anti-phishing*), os *firewalls*, sistemas de detecção de intrusão, entre outras. Contudo, novos artifícios têm sido empregados pelos desenvolvedores de *malware* para dificultar o processo de detecção. Um desses artifícios é proporcionar a alteração do código do *malware* à medida que sua propagação ocorre. Tal técnica, conhecida como "metamorfismo", visa dificultar o processo de detecção por assinatura. Essas versões metamórficas do *malware* são usualmente geradas automaticamente por um componente do código (*engine* de metamorfismo) incorporado no próprio *malware*.

Detecção de *malware* metamórfico é um desafio. O problema com *scanner* baseados em assinatura é que mesmo pequenas alterações no código do *malware* podem conduzir a falhas no processo de detecção e a base de assinaturas requer constante atualização para inserir as variantes recém-criadas. Este trabalho propõe uma metodologia que permite, a partir de um *malware* conhecido, reconhecer variantes metamórficas deste. O método proposto reverte às ações de metamorfismo para obter a versão original e, assim, facilitar o processo de identificação do mesmo. Um processo de comparação é feito visando determinar se o programa testado possui o *malware* buscado. Os resultados alcançados mostram a viabilidade da metodologia proposta, tendo identificado 100% dos *malware* testados sem a ocorrência de falsos positivos.

ABSTRACT

DETECTING METAMORPHIC MALWARE USING CODE NORMALIZATION

Author: Marcelo Freire Cozzolino
Supervisor: Eduardo James Pereira Souto
Co-Supervisor: Flávio Elias Gomes de Deus
Masters Program in Electrical Engineering
Brasília, February of 2012

Malware is a term used to describe all types of malicious software such as viruses, worms or, Trojan horses. To combat these threats, many techniques and tools have been used as anti- (anti-virus, anti-malware, anti-phishing), firewalls, intrusion detection systems, among others. However, new approaches have been used by malware developers to make them more difficult the detection process. One of them is to provide the code change every time it copies itself. This technique is known as "metamorphism" and aims to defeat string signature based detection. These versions of metamorphic malware are usually generated automatically by a component of the code (metamorphic engine) that is incorporated in the malware itself.*

Detection of metamorphic malware is a challenge. The problem with simple signature-based scanning is that even small changes in the malware code may cause a scanner to fail and the signature database requires constant updates to detect newly variants. This work proposes a methodology that allows, from a known malware, recognizing its metamorphic variants. The proposed method reverses the actions of metamorphism for the original version, and thus facilitates the process of identifying the same. A comparison process is done to determine if the tested file has the malware sought. To demonstrate the feasibility, the proposed methodology was applied to set metamorphic variants of a malware, which identified 100% of malware tested without the occurrence of false positives. Moreover, we also compare our approach with commercial antivirus and show that our approach is more effective than existing classification systems.

SUMÁRIO

1. INTRODUÇÃO 1

1.1. MOTIVAÇÃO 2

1.2. OBJETIVO 3

1.3. METODOLOGIA DE TRABALHO 4

1.4. ORGANIZAÇÃO DA DISSERTAÇÃO 4

2. MALWARE 6

2.1. TERMINOLOGIA 6

2.1.1. Vírus de computador 6

2.1.2. Worms 6

2.1.3. Trojan Horse (Cavalo de Tróia) 6

2.1.4. Botnets 7

2.2. ORIGENS HISTÓRICAS DOS MALWARE 7

2.3. A EVOLUÇÃO DOS *MALWARE* 8

2.3.1. Malware Cifrados 9

2.3.2. Malware Oligomórfico 9

2.3.3 *Malware* Polimórficos 9

2.3.3. Malware Metamórficos 10

2.4. ANÁLISE DE MALWARE 10

2.4.1. Procedimento de Análise 12

2.5. TRABALHOS RELACIONADOS 13

2.5.1. Análise da Sequência de Instruções 13

2.5.2. Mineração de Dados 14

2.5.3. Normalização de Código 15

2.6. CONSIDERAÇÕES FINAIS 17

3. METODOLOGIA PROPOSTA 18

3.1. PREPARAÇÃO DA BASE DE REFERÊNCIA 18

3.1.1. Disassembler (Etapa 1) 20

3.1.2. Levantamento do fluxo do programa (Etapa 2) 21

3.1.3. Divisão em *tokens* (Etapa 3) 22

3.1.4. Normalização (Etapa 4) 23

3.1.5. Cálculo de Impressões Digitais dos *tokens* (Etapa 5) 26

3.2. PROCEDIMENTO DE DETECÇÃO 26

3.2.1. Comparação dos Códigos Normalizados 26

3.2.2. Tratamento de Falsos Positivos 27

3.3. CONSIDERAÇÕES FINAIS 28

4. EXPERIMENTAÇÕES PRÁTICAS 29

4.1. AMBIENTE DE EXPERIMENTAÇÃO 29

4.2. ESTUDO DE CASO 29

4.2.1. Normalização 31

4.3. RESULTADOS 33

4.3.1. Avaliação comparativa de antivírus comerciais 34

4.3.2. Avaliação usando a metodologia proposta 37

4.3.3. Análise dos resultados 40

4.4. RESUMO DO CAPÍ|TULO 42

5. CONCLUSÕES E TRABALHOS FUTUROS 43

LISTA DE TABELAS

Página

Ranking dos maiores produtores de *malware* no mundo 1

Atividades maliciosas na América Latina 2

Ordem cronológica da evolução dos *malware* 7

Exemplo de metamorfismos 24

Sequência de instruções normalizadas por substituição 27

Instruções normalizadas por remoção de lixo 28

Resultado do teste do *malware* "W32 Evol.a" no VirusTotal I 29

Resultado do teste do *malware* "W32 Evol.a" no VirusTotal II 30

Pontuação heurística da comparação do "W32 Evol.a" com suas versões metamórficas, separadas por geração de metamorfismo, desfazendo (D) e não desfazendo (ND) o metamorfismo 32

Pontuação heurística de similaridade do "W32 Evol.a" com arquivos livres de contaminação (sem o *malware*), desfazendo (D) e não desfazendo (ND) o metamorfismo 32

Pontuação heurística de similaridade do "W32 Evol.a" com outras versões do "W32 Evol", desfazendo (D) e não desfazendo (ND) o metamorfismo 33

Diferenças máxima e mínima entre a pontuação heurística desfazendo (D) e não desfazendo (ND) o metamorfismo e pontuação heurística máxima e mínima para cada geração do arquivos de *malware*...41

LISTA DE FIGURAS

Página

1. Diagrama do processo de geração de base de *tokens* normalizados 17

2. Levantamento do fluxo do programa 18

3. Divisão em *tokens* 20

4. Exemplo da divisão em *tokens* do código *assembler* 20

5. Diagrama da normalização dos *tokens* 22

6. Diagrama da detecção 23

7. Código do *malware* "W32 Evol a" aberto no Ida Pro 25

8. Exemplo de listagem gerada pelo Ida Pro 25

LISTA DE NOMENCLATURA E ABREVIAÇÕES

Malware	Software Malicioso
Trojan Horse	Cavalo de Tróia
Botnet	Computadores infectados por um tipo específico de *malware*, que oferece ao invasor o controle remoto destas máquinas
AL	América Latina
SCADA	Sistemas de Supervisão e Aquisição de Dados
Op1.	Operação 1
Op2.	Operação 2
Opn.	Operação n
ID	Impressão Digital do *Token*
IDC	Impressão Digital Composta do *Token*
Arquivo Questionado	Arquivo que se deseja testar se há a presença do *malware* procurado

1. INTRODUÇÃO

Nas últimas décadas, a quantidade de softwares maliciosos tem aumentado rapidamente. O termo *malware* (proveniente do inglês *malicious software)* é usado para classificar um *software* destinado a se infiltrar em um sistema de computador alheio de forma ilícita, com o intuito de causar algum dano ou roubo de informações (confidenciais ou não). Vírus de computador, *worms*, *trojan horses* (cavalos de troia) e *spywares* são considerados *malware* (E. Skoudis 2004).

Tais softwares maliciosos são responsáveis por perdas financeiras que ultrapassam centenas de milhões de dólares anuais**.** Por exemplo, em 2006, os prejuízos ocasionados por *worms* foram de aproximadamente US$ 245 milhões, somente entre provedores de acesso norte-americanos (Morin, M. 2006). Segundo (Relatório Anual da Panda Labs, 2009), somente no ano de 2009 surgiram mais novos *malware* do que em todos os anos anteriores. Nesse mesmo ano, o Pentágono gastou com recuperação de danos causados pelos *malware* aproximadamente US$ 100 milhões.

Segundo (Symantec, 2010), o Brasil em 2010 se mantém pelo segundo ano consecutivo como o quarto maior produtor de *malware* do mundo, conforme mostrado na tabela 1.1.

Tabela 1.1 - Ranking dos maiores produtores de *malware* no mundo

Fonte	**2010**		**2009**	
	Posição no *Ranking* Mundial	**Porcentagem geral**	**Posição no *Ranking* Mundial**	**Porcentagem geral**
Estados Unidos	1	19%	1	20%
China	2	16%	2	9%
Alemanha	3	5%	5	5%
Brasil	4	4%	4	5%
Reino Unido	5	4%	3	6%
Índia	6	4%	7	3%
Coréia do Sul	7	4%	9	3%
Itália	8	3%	10	3%
Taiwan	9	3%	6	4%

No contexto de América Latina (AL), o Brasil é o principal produtor de atividades maliciosas, chegando a ter 44% de toda a atividade na AL em 2010. A Tabela 1.2 mostra o quadro de atividades maliciosas na AL em 2009 e 2010.

Tabela 1.2 – Atividades maliciosas na América Latina

Ranking Mundial		**Fonte**	**Ano**	
2010	**2009**		**2010**	**2009**
1	1	Brasil	44%	42%
2	3	México	12%	13%
3	2	Argentina	10%	13%
4	4	Colômbia	7%	8%
5	5	Chile	6%	7%
6	10	Uruguai	4%	1%
7	6	Venezuela	3%	3%
8	7	Peru	3%	3%
9	9	República Dominicana	1%	1%
10	11	Panamá	1%	1%

De acordo com estas informações, o Brasil se apresenta em posição relevante em nível mundial, tanto como vítima quanto como produtor, no que se refere à atividade de *softwares* maliciosos. A participação do Brasil na América Latina aumentou em 2% entre 2009 e 2010, justificando a atenção a este problema.

1.1. MOTIVAÇÃO

Existe uma constante necessidade de aprimoramento das soluções existentes e de desenvolvimento de novas técnicas e ferramentas para fazer frente ao crescimento das ameaças virtuais, como observado nas Tabelas 1.1 e 1.2.

Nesse contexto, para combater tais ameaças, vários produtos comerciais e não comercias tem sido empregados como os anti-* (*anti-virus, anti-malware, anti-phishing*), os *firewalls*, sistemas de detecção de intrusão, entre outros. Contudo, essa disputa de “gato e rato” entre os criadores de vírus e os desenvolvedores de soluções está longe de acabar. Novos artifícios têm

sido empregados pelos desenvolvedores de vírus para dificultar o processo de detecção como a utilização de técnicas de ciframento, metamorfismo segundo (Wing Wong and Mark Stamp, 2006), polimorfismo, entre outras.
Atualmente, a principal técnica de detecção de vírus, utilizada em antivírus comerciais, 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 vírus, os escritores desse código malicioso têm empregado diferentes técnicas de ofuscação (veja o Capítulo 2 para detalhes).

Uma delas possibilita a alteração do código do vírus à medida que sua propagação ocorre. Tal técnica, conhecida como "metamorfismo", visa dificultar o processo de detecção por assinatura. Essas versões metamórficas do *malware* são geralmente geradas automaticamente por um componente do código (*engines de metamorfismo*) que é incorporado no próprio *malware*. Assim, mesmo pequenas alterações no código viral podem conduzir a falhas no processo de detecção ou requerer constantes atualizações na base de assinaturas para inserir as variantes recém-criadas. Como o número de versões metamórficas pode crescer exponencialmente, torna-se praticamente impossível sua detecção usando apenas assinaturas.

Além disso, existem também *malware* não metamórficos, mas com variantes produzidas de forma similar. Neste caso, o atacante faz uso de um programa para produzir as variantes de um vírus. Portanto, a *engine* de metamorfismo não está inserida no próprio código do vírus e sim no programa utilizado para gerar as variantes do vírus.

Na tentativa de simplificar o processo de busca e detecção por *malware* metamórficos, este trabalho propõe a utilização de uma metodologia que permita reconhecer variantes metamórficas de um mesmo *malware*. O objetivo é desfazer as mudanças metamórficas do *malware*, de forma a obter uma versão no estado "original" do *malware*, e fazer comparação destas versões originais para identificar correspondência entre os programas. Neste trabalho, a versão original será denominada de "versão normalizada".

1.2. OBJETIVO

Desenvolver uma metodologia automatizada que permita o reconhecimento de variações "metamórficas" de um mesmo código. A metodologia proposta baseia-se em técnicas de engenharia reversa que tentam desfazer o trabalho de *engines* metamórficas entre os códigos,

de forma a torná-los mais semelhantes e possibilitar a comparação entre trechos de código que se assemelhem ao do *malware* procurado. Para tanto, um tratamento heurístico é empregado para computar o grau de semelhança entre códigos e assim definir se o arquivo analisado é uma variante metamórfica do *malware* ou não.

1.3. METODOLOGIA DE TRABALHO

Para atingir o objetivo propostos foi feito o levantamento bibliográfico para identificação do estado da arte.

Com base nas informações obtidas nos levantamentos, foi definida uma estratégia de abordagem do problema, tanto para realizar a comparação quanto para desfazer as diferenças promovidas pelo metamorfismo. A seguir foi implementada uma ferramenta de *software* para validar esta estratégia.

Para execução de testes foi criada outra ferramenta de *software* responsável por gerar amostras metamórficas de *malware* para estas serem submetidas à metodologia de detecção desenvolvida, possibilitando a identificação de erros e o aprimoramento da ferramenta de detecção.

Por último foi feito teste comparativo de identificação das mesmas amostras por anti-vírus comerciais. Os resultados atingidos são apresentados.

1.4. ORGANIZAÇÃO DA DISSERTAÇÃO

Além da presente introdução, esta dissertação está estruturada da seguinte forma:

- O Capítulo 2 descreve a terminologia utilizada neste trabalho, apresenta um histórico da evolução do vírus e descreve as principais técnicas de ofuscação utilizadas pelos atacantes para dificultar a identificação de *malware*. Além disso, apresenta alguns trabalhos existentes e que foram utilizados como referência para o desenvolvimento da metodologia proposta. Essa apresentação é agrupada em três categorias: trabalhos que consideram a análise da sequência de instruções executadas pelo *malware*; trabalhos que empregam alguma técnica de mineração e/ou aprendizagem de máquina; e

trabalhos que focam em realizar alguma “normalização” do código, para facilitar o trabalho de antivírus na identificação de variantes de *malware* metamórficos.

- O Capítulo 3 descreve os principais passos da metodologia proposta para detecção de *malware* metamórficos. São apresentados detalhes sobre o processo de construção de uma base de dados de referência do *malware* alvo. Esse processo é composto por cinco etapas: o *disassembler* do código binário; a divisão das instruções do programa em células pequenas (divisão em *tokens*); o processo de levantamento do fluxo do programa que um *token* pode tomar; o processo de normalização de *tokens* para efeito de comparação; e o cálculo de impressões digitais a partir dos *tokens* alvos.

- O Capítulo 4 apresenta uma avaliação da metodologia através de um estudo de caso utilizando o vírus metamórfico “W32 Evol”. Além disso, são mostrados os resultados numéricos considerando 63 versões do *malware* metamórficas do vírus Evol. Adicionalmente, também é realizado um estudo comparativo dos resultados obtidos pela solução e resultados de outras soluções comerciais.

- O Capítulo 5 apresenta algumas conclusões obtidas com o desenvolvimento do trabalho e comenta sugestões de melhorias e trabalhos futuros.

2. MALWARE

Este Capítulo descreve conceitos necessários ao entendimento deste trabalho, um breve histórico mostrando a evolução dos vírus e das técnicas de ofuscação utilizadas pelos atacantes para dificultar a identificação dos *malware*. Além de trabalhos que apresentam contra-medidas a estas técnicas, para serem usadas por desenvolvedores de anti-vírus.

2.1. TERMINOLOGIA

Nesta seção serão apresentadas algumas definições utilizadas neste trabalho. De acordo com (Peter Szor, 2005), a nomenclatura usada para definir os diversos tipos de programas maliciosos é descrita nas seções a seguir:

2.1.1. Vírus de computador

É um programa destinado a alterar o sistema vitimado sem a autorização do usuário, segundo (The McGraw-Hill Companies, 2010). Para atingir esse objetivo, ou como efeito colateral, o vírus pode copiar a si próprio em múltiplas localidades ou em outros computadores em uma rede, modificar objetos de sistema no disco ou memória, corrompendo o funcionamento normal do sistema operacional.

2.1.2. Worms

Assim como um vírus, cria cópias de si mesmo de um computador para outro, mas faz isso automaticamente (Eilam 2005). *Worms* tem a capacidade de se disseminar rapidamente em um breve espaço de tempo, manipulando vulnerabilidades de um sistema operacional ou de um aplicativo.

2.1.3. Trojan Horse (Cavalo de Tróia)

É um *malware* que age como a lenda do cavalo de Troia, entrando no computador e liberando uma porta para uma possível invasão. Os *trojans* atuais são disfarçados de programas

legítimos (por exemplo, um jogo), embora, diferentemente de vírus ou de *worms*, não criam réplicas de si.

2.1.4. Botnets

Segundo (Stone, Cova, Cavallaro, Gilbert, Szydlowski, Kemmerer, Kruegel, Giovanni, 2009), são redes de máquinas comprometidas e controladas remotamente por um atacante. Os computadores em uma rede *botnet* foram infectados por um software, que permite ao invasor, comandar os computadores à distância. Estas redes de computadores zumbis são utilizadas, na maioria das vezes, com fins lucrativos ou políticos.

2.2. ORIGENS HISTÓRICAS DOS MALWARE

A cada ano, os *malware* tornaram-se cada vez mais perigosos, a despeito de todas as estratégias e ferramentas que foram e vem sendo propostas para prevenir e controlar sua difusão. Esta seção apresenta uma breve ordem cronológica da evolução dos *malwar*e.

Tabela 2.1 – Ordem cronológica da evolução dos *malwar*e

Ano	Descrição
1971	O vírus "*Creeper*" se replicou na rede ARPANET, uma precursora da Internet. Foi feito por Bob Thomas para demonstrar a viabilidade de um programa auto-replicante. Era inofensivo, apenas apresentava uma mensagem na tela do computador infectado "Eu sou o Creeper, apanha-me se és capaz!".
1972	Veith Risak publicou um artigo no qual descreve um vírus funcional, totalmente escrito em linguagem *assembler* para computador SIEMENS 4004/35.
1980	Jürgen Kraus escreve sua tese, "programas auto-replicantes", na qual compara programas de computador auto-replicantes a vírus biológicos.
1982	Richard Skrenta fez o primeiro vírus (Elk Cloner) que se propagava de forma descontrolada. Propagava-se através de *floppy disks* (disquetes). Era inofensivo, somente escrevia uma mensagem, com objetivo de diversão. Executava no Sistema Operacional Apple DOS 3.3.
1984	Fred Cohen escreve um artigo onde pela primeira vez chama um programa auto-replicante de "vírus".
1986	Os irmãos paquistaneses Basit e Amjad Alvi desenvolveram o primeiro vírus para PC, chamado *Brain*. Segundo os autores, o objetivo era deter a pirataria de um software de sua autoria.
1987	Surgiu o vírus Jerusalém, sendo o primeiro vírus a ter um impacto global. Em cada 6ª feira, dia 13, este vírus apagava os programas que estivessem em funcionamento no computador. O nome deve-se ao facto de ter sido detectado pela primeira vez na Universidade Hebraica de Jerusalém.
1988	O *worm* feito por Robert T. Morris foi o primeiro a realmente se espalhar com sucesso na *internet*. A partir do início dos anos 90 a *internet* passou a ser o principal meio de propagação dos *malware* (Spafford, 2003).

1992	O vírus Michelangelo acordava no dia 06 de março (dia de nascimento de do artista renascentista) e apagava informação essencial do computador e infetava discos rígidos.
1999	O *malware* "Melissa" é o primeiro a ter sucesso se propagando através de mandar a si mesmo por e-mail (Zelonis, 2003). Devido ao sucesso de propagação, Melissa é considerada o primeiro vírus massivo via e-mail. Este *malware* infetava os ficheiros Microsoft Word e enviava-se a si próprio através do Outlook.
2000	A era um vírus tipo worm (que se auto replica) que afetou dezenas de milhões de computadores pessoais iniciou com o malware "I love you" agia como o Melissa porém enviava também senhas e usuários das máquinas infectadas para o autor do *malware* (Zelonis, 2003).
2001	Surgia um dos mais famosos worm, o Code Red. Numa semana atacou 400 mil servidores web (ISS Web Microsoft). Este *worm* substituía a página inicial dos sítios web com a mensagem "*Hacked by chinese*".
2004	Sasser infetou mais de um milhão de computadores e provocou prejuízos de 18 mil milhões de dólares. Este *worm* explorava uma vulnerabilidade do Microsoft Windows XP e 2000 para se espalhar de uma forma muito rápida.
2005	Iniciou a era dos *Botnets* (agentes de software que funcionam autonomamente). O My Tob era um *worm* que combinava as características de zombie (programa controlado remotamente) e de envio massivo de mensagens de correio eletrônico. Com My Tob, os vírus tornaram-se um negócio para desenvolver atividades de espionagem (spyware), difusão de correio indesejado (*spam*), hospedagem nos servidores de conteúdo indesejado, entre outras (Info Security, 2011).
2007	A rede Storm (controlada remotamente) foi propagada usando um *Trojan Horse* inseridos em *emails* (*spams*). Estima-se que o *botnet* Storm tenha infectado de um milhão a 50 milhões de computadores (InformationWeek, 2007)
2009	O *worm* Conficker, também conhecido como Downup, Downadup, e Kido, infectou mais de 7 milhões de sistemas em mais de 200 países. Conficker inicialmente se espalhou explorando uma vulnerabilidade no sistema operacional Windows. Versões subsequentes (por exemplo, Conficker.c) também se espalham através de drives USB e sistemas de arquivos de rede (G. Lawton, 2009).
2010	Inicio da guerra cibernética com o *worm* Stuxnet, projetado para atacar o sistema de controle industrial SCADA (Controle de Supervisão e Aquisição de Dados), usado para controlar e monitorar processos industriais (New York Times, 2010).

2.3. A EVOLUÇÃO DOS *MALWARE*

Os desenvolvedores de *malware*, no intuito de dificultar sua identificação, usam diferentes técnicas para evitar que sejam detectados por soluções de segurança baseadas em assinaturas. Técnicas que buscam dificultar a identificação e/ou interpretação de código são chamadas técnicas de ofuscação. De acordo com (Peter Szor,2005), técnicas de ofuscação estão intrinsecamente atreladas à evolução do código dos *malware* e são descritas nas próximas seções.

2.3.1. Malware Cifrados

Um dos métodos avançados que os produtores de *malware* usam para esconder seus códigos é cifrar o corpo com chaves diferentes. Assim, o *malware* será diferente a cada propagação. Entretanto, tal técnica de ofuscação ainda é vulnerável a técnica de busca por assinatura devido ao fato da sub-rotina que executa a cifragem não poder ser cifrada, podendo assim ser usada como assinatura. Além disso, muitos dos atuais programas de antivírus são capazes procurar pelo código do *malware* depois de terem sido carregados na memória (Computer vírus, 2011).

2.3.2. Malware Oligomórfico

Malware Oligomórfico é uma variação do cifrado na qual o *malware* varia o algoritmo de decifragem. Um modo simples de se implementar é utilizar várias *engines* de cifragem e escolher uma aleatoriamente durante a o processo de propagação. Assim, a maior parte das *engines* poderá permanecer cifrada e só a correntemente usada precisará estar exposta. Isso torna o algoritmo de busca pela assinatura da *engine* de decifração mais difícil (Venkatesan, 2008).

2.3.3 *Malware* Polimórficos

Código polimórfico foi a primeira técnica que representava um aumento grande de complexidade na identificação de vírus. Assim como no *malware* cifrado, um vírus polimórfico infecta arquivos com uma cópia cifrada de si mesmo, que é posteriormente restaurada à sua forma original por um módulo de decodificação. No caso do *malware* polimórfico, no entanto, este módulo de decodificação também é modificado em cada infecção. Um *malware* polimórfico bem escrito não tem partes que permanecem idênticas entre infecções, o que torna muito difícil de detectar diretamente usando assinaturas. Técnicas de detecção de vírus polimórfico são usualmente feitas através da execução do código deste em um emulador de sistema operacional ou pela análise de padrões estatísticos do corpo do *malware* cifrado. Para ativar o código polimórfico, o *malware* tem que ter uma *engine* polimórfica em algum lugar em seu corpo cifrado.

Alguns *malware* utilizam código polimórfico de forma a restringir a taxa de mutação do vírus significativamente. Por exemplo, um vírus pode ser programado para se transformar apenas ligeiramente ao longo do tempo ou pode ser programado para se abster de mutação quando

infecta um arquivo em um computador que já contém cópias do *malware*. A vantagem de usar o código lento como polimórfico é que ele torna mais difícil a obtenção de amostras representativas do *malware*, uma vez que os arquivos de isca que são infectados normalmente contêm amostras idênticas ou semelhantes do *malware*. Isto irá tornar mais provável que a detecção pelo antivírus não será confiável e, em alguns casos, o *malware* poderá ser capaz de evitar a detecção (Computer vírus, 2011).

2.3.3. Malware Metamórficos

Malware polimórficos têm um ponto fraco: o corpo principal do vírus é idêntico em cada geração. Portanto, se um *malware* polimórfico é de alguma forma decifrado, pode posteriormente ser detectado baseado em padrões de detecção. *Malware* metamórfico é a próxima etapa da evolução de *malware*, uma vez que não dependem do processo de cifragem como uma técnica de dissimulação. O *malware* metamórfico transforma a estrutura do seu próprio código através de operações como inserção de código inerte, troca de registradores (usa registradores diferentes entre as gerações do *malware*), reordenação do código (quando a ordem não tiver influência na execução do programa, troca a posição de instruções ou de blocos de código), troca de instruções por instruções equivalentes e ofuscação de controle de fluxo (dividindo um bloco continuado de programa em vários blocos em localizações diferentes, mas mantendo o mesmo fluxo ligando os blocos com instruções de desvio). Em suma, produzindo variantes muito diferentes, porém com as mesmas funcionalidades.

Diferente das modalidades de *malware* cifrados já descritas anteriormente neste trabalho, os metamórficos permanecem diferentes não só em disco, mas também na memória do computador (InformationWeek, 2007).

2.4. ANÁLISE DE MALWARE

Uma técnica normalmente usada no combate a *malware* é a engenharia reversa. De acordo com (Eilam, 2005), engenharia reversa é um conjunto de técnicas e ferramentas para entender o funcionamento de um software. É um processo de análise de um sistema para identificar seus componentes e suas inter-relações, para criar representações do sistema em um nível

mais alto de abstração. No caso dos *malware*, é aplicada na interpretação do código de máquina do *malware* para tentar entender seu funcionamento.

A engenharia reversa pode ser feita de forma manual, através de uma análise feita por um programador, ou de forma automatizada, quando um *software* tenta identificar alguma estrutura do *malware*. Normalmente isso é feito para identificar a presença de um *malware* em um programa.

Segundo (Handle, 2010), existem diferentes abordagens de engenharia reversa e definir qual abordagem mais apropriada depende da aplicação alvo, da plataforma em que roda a aplicação, onde foi desenvolvida e qual tipo de informação deseja extrair. As abordagens se dividem em dois principais tipos de metodologias: análise estática e análise dinâmica.

A análise estática de programas é uma técnica utilizada para coletar informações sobre o programa sem a necessidade de executá-lo. Envolve o uso de um *disassembler* que faz a conversão do código binário para código *assembly*, cujo formato é mais compreensível por um humano do que uma sequência de bits. A análise estática exige um melhor entendimento do programa (comparado com a análise dinâmica), pois o fluxo da execução é somente baseado no código do programa. Ferramentas de análise de fluxo de controle coletam informações para uma melhor compreensão sobre o fluxo de execução do programa. Como o fluxo pode ser dependente dos valores contidos em posições de memória, a análise do fluxo de dados prediz o conjunto de valores que determinada memória pode vir a assumir durante a execução do programa. Outra ferramenta que se enquadra na análise estática são os decompiladores que tentam reverter o processo de compilação para produção de um código de alto nível. Contudo, na maioria das arquiteturas, a recuperação do código de alto nível não é realmente possível, pois existem elementos significativos que são removidos durante a compilação uma vez que o âmbito deste é otimização e não a legibilidade do código (InformationWeek, 2007).

A análise dinâmica de programas é uma técnica onde a coleta de informações é feita em tempo de execução diante de uma determina entrada. Nesta análise, é possível observar como a entrada interage como o fluxo de execução e os dados do programa. A análise basicamente envolve o uso de um depurador (*debugger*) que permite a um analista observar o programa durante sua execução. Um depurador possui tipicamente duas características básicas: habilidade de ajustar pontos de parada (*breakpoints*) no programa e capacidade de verificar o estado atual do programa (registradores, memória, conteúdo da pilha). A análise dinâmica também pode ser feita através de emulação, onde o código é executado em um emulador que

provê funcionalidades similares ao sistema original, assim como um arcabouço para auxiliar um analista no entendimento do código. Técnicas de *profiling* e *tracing* também são consideradas como dinâmicas. A primeira é utilizada para contagem dos números de execuções, ou a quantidade de tempo despendida, de diferentes partes do código; a segunda, em vez de realizar somente a contagem, registra também quais partes do código do programa foram executadas (InformationWeek, 2007).

2.4.1. Procedimento de Análise

O primeiro passo para aplicação de engenharia reversa é o *disassembler*, onde o código binário é transformado em um código em linguagem de máquina (*assembler*). Contudo, o processo de *disassembly* não é ciência exata. Ferramentas de análise estática de programas que dependem de sua saída podem gerar resultados incorretos. Tais ferramentas são responsáveis pela abstração dos procedimentos, geração do grafo de fluxo de controle, análise do fluxo de dados e decompilação. Os *disassemblers* podem ser genericamente divididos em dois tipos: varredura linear e transversal recursivo (InformationWeek, 2007).

Os *disassemblers* de varredura linear começam pelo processamento do segmento de texto (.text) da imagem do código binário obtido no cabeçalho do arquivo. O processo de *disassembly* é realizado sequencialmente até atingir o final do segmento de texto (InformationWeek, 2007). Contudo, isso pode gerar erros de interpretação. Linguagem *assembly* tem a característica de ter instruções misturadas com parâmetros. Se o ponto inicial de interpretação de uma instrução estiver errado pode-se interpretar um parâmetro como instrução e, como consequência disso, fazer uma sequência de interpretações erradas no código. A inserção proposital de instruções inúteis em pontos estratégicos provocam erros de interpretação em parte do código. Entretanto, (Cullen Linn, 2003) ressalta uma propriedade interessante do *assembly*, mesmo quando é feita a interpretação errada do ponto de início de uma instrução, em poucas instruções ocorrerá à auto-sincronização do *disassembler*, o que o faz voltar a interpretar corretamente. Isso ocorre muito rapidamente, em média em uma a duas instruções. Um programa que queira ofuscar o código usando esta dificuldade de interpretação do *assembly* tem que levar em conta essa particularidade.

Um *disassembler* transversal recursivo visa superar este problema fazendo com que o processo de *disassembly* acompanhe do fluxo de controle do programa. Caso haja dados no meio do fluxo de instruções, o *disassembler* acompanha o salto, e consequentemente, não interpreta erroneamente os dados. Contudo, o processo de *disassembly* é mais complexo, uma

vez que tem que deduzir os possíveis destinos estaticamente, o que nem sempre é uma tarefa trivial como no caso dos desvios indiretos. Em desvios indiretos, o endereço para o qual o programa será desviado depende do conteúdo de registradores ou posições de memória. Saber esse conteúdo em análise estática pode ser bem trabalhoso, pois é necessário acompanhar o passo a passo do programa para saber o seu valor. Pode ser que esse valor tenha sido carregado poucas instruções antes ou pode ser que esta carga esteja bem distante, e pode ser também que esse valor seja resultado de diversas operações.

Técnicas de ofuscação de código que dificultam o processo de *disassembler* podem ser usadas pelos criadores de *malware* (vírus, *worms*,..), causando grandes dificuldades a programas que buscam interpretar automaticamente o código do *malware*.

2.5. TRABALHOS RELACIONADOS

Esta seção descreve um conjunto de trabalhos relacionados ao processo de identificação de *malware*. O objetivo é apresentar uma visão geral sobre abordagens propostas, incluindo tanto as baseadas em análise de código quanto as que focalizam outras evidências deixadas em um sistema com suspeita de contaminação. Todas as abordagens possuem um conjunto particular de características individuais associadas à forma como cada uma trata a tarefa a qual se destina, oferecendo uma visão mais ampla dos desafios associados a esta tarefa.

2.5.1. Análise da Sequência de Instruções

A proposta de (Zhang, 2008) assume que um *malware* deve fazer uso de chamadas ao sistema operacional, com o objetivo de diminuir o tamanho de seu próprio código. A técnica proposta consiste em uma análise estática do código do programa investigado, em busca de chamadas ao sistema e, com base em uma sequência específica de chamadas, fazer uma “impressão digital” do código a ser utilizada como base de um processo de identificação de contaminações em outros sistemas. As principais limitações desta abordagem se originam do fato que o significado de uma chamada a sistema pode depender do valor de registradores no momento da chamada, exigindo que se saiba o estado de cada registrador a cada momento. Outras limitações são a dificuldade de se determinar qual função está sendo chamada e

identificar uma chamada se esta for feita de forma indireta (chamadas cujo endereço da rotina é lido de uma posição de memória ou registrador).

Já (Batista, 2008) trabalha com a extração de características dos códigos executáveis de *malware*, através da utilização de técnicas de inteligência artificial e análises estatísticas do código, para identificar quais características seriam comuns a vírus e a não vírus e, pela presença de mais de uma delas, classificar programas suspeitos como vírus ou não. Entretanto, o autor oferece poucas informações a respeito dos critérios de escolha das características que são relevantes para este processo.

2.5.2. Mineração de Dados

Schultz et. al. (2001) propõe uma abordagem que, através do uso de inteligência artificial e uma base de exemplos de programas que são e que não são *malware*, busca fazer uma identificação automatizada de um determinado programa, através da classificação de estruturas que são comuns/incomuns em *malware* e da presença ou ausência dessas estruturas em programas suspeitos. A técnica proposta obtém um bom resultado de identificação, porém com alto grau de falsos positivos, devido à inclusão de estruturas que também podem ser comumente encontradas em programas normais.

Wong e Stamp (2006) apresentam uma técnica de reconhecimento de padrões conhecida por "cadeias de Markov escondidas" nas instruções do código de máquina de um software, buscando identificar se o mesmo é ou não um *malware*. O algoritmo foi testado a partir de cinco ferramentas de geração automática de *malware*. Como resultado, o algoritmo foi eficiente em identificar um *malware* proveniente de alguma das ferramentas de geração com o qual foi treinado, mas não para identificar um *malware* desenvolvido de outra maneira.

Lakhotia e Chouchane (2006) propõe uma metodologia para identificar programas feitos pelo mesmo kit de criação de vírus ou pela mesma *engine* metamórfica. Conhecendo as transformações feitas por determinada *engine*, eles buscam sequências de instruções características de terem sido geradas por ela, que funcionam como sendo uma "assinatura" da *engine*. Usam um critério de pontuação heurística para validar o número de sucessos e poder dar um parecer. Obtiveram resultados muito bons quando a *engine* é idêntica ou tem diferenças inferiores a 10% das transformações.

Rieck et al. (2008) demonstram uma técnica de aprendizado e classificação do comportamento de *malware* baseado no fato que famílias de *malware* compartilham padrões de comportamento típico, associados às suas origens e a seu propósito, utilizando estas características não apenas para identificá-los, mas também para classificados distintamente, em função de seu comportamento. O processo proposto compreende as etapas de:

1) Aquisição de dados, onde *malware* são coletados através de *honeypots*;
2) Monitoramento de Comportamento, onde se observa o comportamento e as modificações induzidas pelo *malware* em um ambiente controlado;
3) Extração das características, baseados em relatórios gerados na fase de monitoração;
4) Aprendizado e Classificação, onde são utilizadas técnicas de aprendizado de máquina para enquadrar e classificar o *malware* analisado em uma família específica; e
5) Explanação, onde os modelos gerados são usados para procurar uma correlação entre os comportamentos das diferentes famílias.

Uma das limitações desta proposta é sua incapacidade de simular todo o espectro de possibilidade durante a execução do *malware* dentro do ambiente controlado. Por exemplo, ações associadas a datas específicas, passando pela dificuldade de classificar um *malware* que esteja imitando o comportamento apresentado por outra família, o que levaria a uma classificação equivocada. Finalmente, este modelo ainda não é capaz de identificar novas famílias de *malware* que ainda não tiveram seu comportamento classificado anteriormente.

2.5.3. Normalização de Código

Christodorescu (2007) propõe a "normalização" do código, para facilitar o trabalho de antivírus na identificação de variantes de *malware* metamórfico, se concentrando em tratar três tipos específicos de mutações metamórficas: a) Inserção de código inerte – insere instruções inúteis, mas que não interfere com o desenrolar do programa; b) Mudança de ordenação – muda a ordem de instruções que não alterem o funcionamento final do programa; c) Compactação – a compactação deixa o código em um estado que ocupa menor espaço para só ser descompactado no momento que for ser executado, o que dificulta a identificação do código conhecido devido à impossibilidade de saber qual algoritmo de compactação foi usado. O processo de normalização descrito inicia por reverter a compactação para obter um código executável para depois eliminar os desvios (*jumps*), que alteram a ordem das instruções, gerando um código que representa o fluxo "normal" das instruções. Em seguida

ocorre a eliminação das instruções inertes para assim gerar um novo código que se aproxime ao máximo do código original, que então será submetido à avaliação por processos tradicionais de detecção por assinatura. As principais limitações deste trabalho estão associadas à impossibilidade de garantir que o fluxo original das instruções possa sempre ser completamente reconstruído e que restem somente as instruções realmente relevantes para o funcionamento do *malware* (Christodorescu,2007).

O trabalho de (Bruschi, 2007), baseia-se no fato de que a maioria das transformações utilizadas por *malware*, visando disfarçar sua presença, tem como efeito colateral um aumento no tamanho código binário produzido. Assim, o código resultante pode ser encarado como um código não otimizado, já que contará com alguns comandos irrelevantes, destinados exclusivamente a mascarar o código original. Desta forma, a normalização deste código, através de técnicas de otimização utilizadas comumente por compiladores, terá como efeito a eliminação deste código "inútil", facilitando o processo de reconhecimento. Esta abordagem se diferencia das outras por adicionar uma etapa de "interpretação de código", onde as instruções *assembler* são simplificadas ainda mais em uma linguagem com poucos operadores semânticos. Este código é que será tratado pela normalização e a comparação, na busca da identificação do *malware*. Entretanto, apesar do processo de normalização aplicado durante a execução de testes apresentar tempos razoáveis, quando aplicado a programas de tamanho maior demandou uma grande quantidade de tempo e recursos computacionais, o que limita sua efetiva aplicabilidade (Bruschi, 2007).

Finalmente, (Kim e Moon, 2010), apresentam um mecanismo de detecção de *malware* embutidos em códigos *script*, escritos em linguagens como Visual Basic Script e JavaScript. Como estas linguagens são interpretadas, este tipo de *malware* se propaga em forma de código não compilado ou código fonte. Entretanto, mesmo nesta forma de apresentação, estes códigos estão sujeitos à utilização de técnicas de polimorfismo. Para lidar com isto, o código é analisado e transformado num grafo de dependência que representa as relações de dependência entre as linhas de código semântico, com o foco na manipulação das variáveis utilizadas neste programa. Cada vértice do grafo gerado representa uma linha de código e a dependência entre duas linhas é modelada como uma aresta direcional. O grafo é então normalizado, para eliminar vértices ou conjuntos de vértices que não possuam qualquer relação com os outros vértices do grafo principal. Isto, além de diminuir o tamanho do grafo que será analisado, tem o efeito de eliminar código inserido para efeito de ofuscação. A partir daí, o processo de detecção é tratado como o problema de encontrar o maior isomorfismo

entre um grafo que modele um código previamente identificado com *malware* e um subgrafo do grafo de dependência que modela o código script contaminado. Como encontrar o máximo isomorfismo de um subgrafo é um problema NP-Difícil, a maior limitação deste trabalho é o custo computacional associado a esta atividade, o que poderia gerar tempos de execução impraticáveis (Kim e Moon, 2010).

No trabalho de (Andrew et al., 2006), os autores lidam com a dificuldade de reconhecer variantes metamórficas de um trecho de código e propõe que antes de tentar a comparação entre dois códigos, ambos sejam passados por um processo de "normalização" que faça com que os dois se assemelhem. Este trabalho assume ainda que o número de *engines* de metamorfismo existentes deve ser limitado, devido à dificuldade de implementação, e que provavelmente vários *malware* usaram *engines* similares. Portanto, a criação de *engines* de reversão do metamorfismo para um *malware* deve apresentar bons resultados com muitos outros. Apesar de ressaltar a impossibilidade de garantir qual seria realmente o código original, isso não seria impeditivo para se identificar *malware* de uma mesma origem. Mesmo assim, a maior limitação desta proposta é que qualquer *engine* não tratada poderia ludibriar a ferramenta de detecção.

2.6. CONSIDERAÇÕES FINAIS

Neste capítulo foram apresentados conceitos necessários para entendimento deste trabalho, apresentada uma retrospectiva histórica da evolução do *malware* e descritos trabalhos relacionados já desenvolvidos. Os trabalhos relacionados forma divididos em três famílias: Análise da Sequência de Instruções, Mineração e normalização de código.

3. METODOLOGIA PROPOSTA

Este capítulo apresenta a metodologia proposta para a detecção de *malware* metamórficos, baseada na comparação de códigos normalizados a partir de uma instância de um *malware* ofuscada por operações de metamorfismo. A metodologia deste trabalho é da família da normalização de código, descrita em 2.5.3, buscando desfazer as mutações metamórficas, contida em uma biblioteca conhecida de transformações metamórficas, antes de proceder à comparação. O processo de detecção de um *malware* é composto basicamente pela construção de uma base de referência do *malware* buscado, e usá-la na comparação com o código normalizado de um arquivo sobre questionamento. A metodologia não tratará criptografia ou compactação sendo necessário usar conjuntamente com outros métodos para atenderem estes casos. Este funcionamento conjunto pode ser também feito dinamicamente na memória no caso de dificuldade em se obter o código decifrado em análise estática.

3.1. PREPARAÇÃO DA BASE DE REFERÊNCIA

Antes de iniciar o processo de detecção, é necessária a construção de uma base de dados de referência do *malware* buscado, constituída de um conjunto de elementos normalizados que servirá de referência para as comparações com programas suspeitos de contaminação. Esta base é formada por *tokens* normalizados, que modelam os elementos essenciais de trechos de códigos do *malware*, de forma a registrar a relevância semântica das operações executadas, independente da forma como foram codificadas.

O processo de geração dos *tokens* normalizados é composto por cinco etapas, a saber:

- Etapa 1 – *Disassembler* – Inicia com o processo de *disassemble*r do código binário, ou seja, o código de máquina do programa é convertido em linguagem *assembly*, que possibilitará a interpretação das instruções;
- Etapa 2 – Levantamento do Fluxo do programa – Nesta etapa são levantados quais são os possíveis *tokens* seguintes a serem executados pelo programa, ou sinalizado a impossibilidade de se obter esta informação de forma trivial, em análise estática. Isto é feito para atender as necessidades do algoritmo de comparação. Este levantamento é

restrito a dois caminhos alternativos, por se julgar que é suficientemente representativo para caracterizar a estrutura funcional do *malware*;

- Etapa 3 – Divisão em *tokens* – As instruções do programa são separadas em pequenas células (*tokens*), que serão as unidades funcionais usadas nas etapas seguintes, e também na comparação com o programa questionado. Neste trabalho, o termo programa questionado será usado como referência a programas suspeitos de contaminação por um malware;
- Etapa 4 – Normalização – Os *tokens* passarão um processo que visa tornar semelhantes *tokens* derivados, pelo metamorfismo, de um mesmo *token* inicial;
- Etapa 5 – Cálculo de Impressões Digitais dos *tokens* – É feito o cálculo da Impressão Digital (ID), *hash*, correspondente a cada *token* e da Impressão Digital Composta (IDC) que é formada pelo conjunto da ID do *token* e as IDC dos seus dois *tokens* subsequentes (ou zero quando o *token* subsequente não tiver sido determinado).

A Figura 3.1 apresenta o diagrama do processo de geração dos *tokens* normalizados. Todas estas etapas serão detalhadas nas seções seguintes.

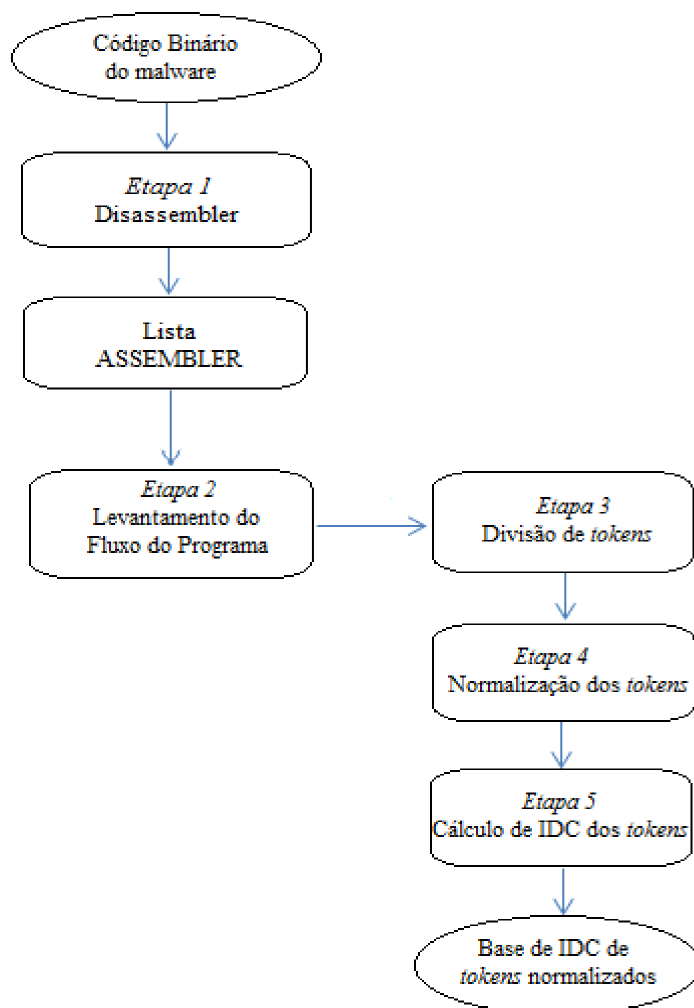

Figura 3.1-Diagrama do processo de geração de base de *tokens* normalizados

3.1.1. Disassembler (Etapa 1)

A base das ferramentas de análise de programas maliciosos ofuscados é o *disassembler* cuja função é a engenharia reversa de um código binário para recuperar um código *assembler*. Este processo pode ser realizado através de *disassemblers* convencionais encontrados no mercado. OllyDbg (OllyDbg, 2011) e IDA Pro (Hex-Rays, 2011) são exemplos de *disassemblers* compatíveis com o Windows, enquanto que Bastard Disassembler (SorceForge, 2011) e LIDA (2004, Mario Schallner) são exemplos de disassemblers compatíveis com o Linux.

3.1.2. Levantamento do fluxo do programa (Etapa 2)

Após o processo de *disassembler* do código inicia-se a etapa que busca levantar a ordem de execução das instruções para construir o fluxo do programa. Para tornar isso possível, são levantadas, para cada instrução, quais as possíveis instruções subseqüentes. Cada instrução terá dois endereços de saída, que serão representados pelas variáveis F1 e F2. F1 será sempre o endereço da instrução seguinte pelo posicionamento da memória e F2 será o endereço para o qual se vaiem caso de desvio. Instruções que permitam mais de dois destinos de saída (retorno de subrotina ou desvios indiretos) não terão o fluxolevantado neste trabalho, devido à complexidade desta operação em análise estática. Nestes casos, os valores de F1 e F2 serão igualados a zero, para sinalizar que o fluxo não foi levantado. Nas chamadas de subrotinas será considerado que a subrotina terá o comportamento padrão, ou seja, retorna para a instrução seguinte. Neste levantamento não será revelado o desvio do programa para dentro da subrotina. A Figura 3.2 apresenta o algoritmo usado no levantamento do fluxo do programa.

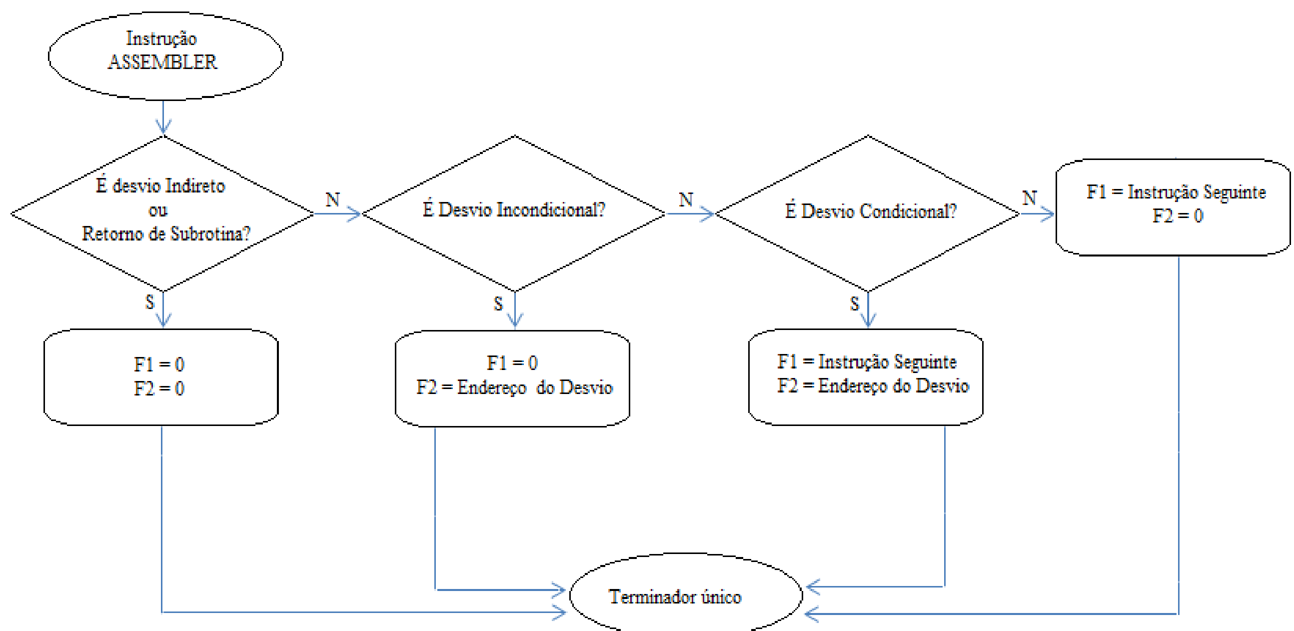

Figura 3.2 – Levantamento do fluxo do programa

3.1.3. Divisão em *tokens* (Etapa 3)

A próxima etapa consiste na divisão do código em partes menores que serão usadas na fase de comparação dos códigos. Essas partes menores terão sua função semântica dentro do código preservada durante as operações de metamorfismo, passando a serem identificadas como *tokens*. A divisão do código nesses *tokens* deve ser feita de forma a que os metamorfismos mantenham o código alterado dentro do mesmo *token*. Ou seja, o metamorfismo não mudará o número de *tokens* e, caso uma nova divisão em *tokens* for feita em um mesmo código metamorfizado, cada *token* permanecerá apenas com códigos metamorfizados oriundos do *token* original. Metamorfismos podem além de alterar a forma de apresentação dos *tokens*, alterar a ordem dos *tokens* em que essa ordem não seja importante.
Para a divisão dos *tokens* foram utilizados critérios para atender as seguintes condições:

1. Os *tokens* devem ser executados a partir da primeira linha e terminados na última linha. O objetivo é evitar que o código de um *token* começasse a ser executado por uma linha que não a primeira deste, e que só saísse dele pela última instrução. Isso faz com que em *tokens* equivalentes, códigos também equivalentes tenham sido executados;

2. A divisão dos *tokens* deve facilitar qual(i)s seria(m) o(s) possível(is) *token(s)* a ser(em) executado(s) logo após a saída do token corrente. Caso não seja possível identificar de forma trivial em análise estática qual o *token* seguinte, este fato deve ser sinalizado.

Foram usados três tipos de delimitadores dos *tokens*: as linhas assembler com rótulos (*labels)*, as instruções de desvio, e as instruções de retorno de subrotina. Através do fluxo de programa, levantado pela última instrução do *token,* serão identificados *tokens* seguintes, o que será usado posteriormente na etapa de comparação. A Figura 3.3 representa o diagrama do processo de divisão em *tokens*. A Figura 3.4 mostra um exemplo de código *assembler* dividido em Tokens, mostrando também os dois possíveis *tokens* seguintes.

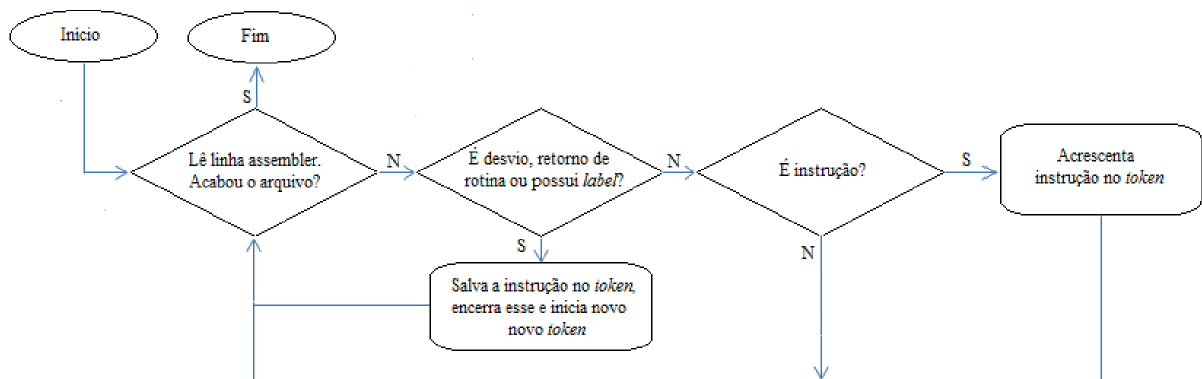

Figura 3.3 – Divisão em *tokens*

Token	endereço	instrução	Parâmetros
36	4011f2h	cmp	eax, 77F00000h
36	4011f7h	jnz	short 401221h
Próximo *token* 1 = 37			Próximo *token* 2 = 40
37	4011f9h	mov	dword ptr [esi], 550004C2h
37	4011ffh	mov	dword ptr [esi+4], 0C244C8Bh
37	401206h	jmp	short 401221h
Próximo *token* 1 = 40			Próximo *token* 2 = 40
38	401208h	cmp	eax, 77E00000h
38	40120dh	jnz	short 401261h
Próximo *token* 1 = 39			Próximo *token* 2 = 46
...	...	...	...
40	401221h	mov	edi, eax

Figura 3.4 – Exemplo da divisão em *tokens* do código *assembler*

3.1.4. Normalização (Etapa 4)

A etapa de normalização é responsável por desfazer as alterações que foram implementadas pela *engine* metamórfica, de forma a se obter um código mais próximo ao "original". Existem mudanças metamórficas que só ocorrem em uma direção: Um código após vários metamorfismos consecutivos dificilmente voltará à condição original. Contudo, ainda assim é possível especular qual era o código inicial. Além disso, existem metamorfismos que podem ser executados e, após vários metamorfismos sobrepostos, voltar à condição inicial. Nesse

caso, não é possível especular qual seria o código inicial. Em casos assim, arbitra-se um código como sendo o inicial em todas as ocorrências, de forma a levar todas as variações metamórficas até um ponto comum.

A metodologia proposta se baseia em bibliotecas de metamorfismos de *engines* conhecidas, buscando desfazer uma a uma as mutações. A ordem em que serão desfeitas pode alterar o resultado, pois, para garantir a volta ao código inicial, seria necessário desfazer os metamorfismos na ordem em inversa em que foram feitos. Não existe ordem em que se desfaça os metamorfismos que se garanta conseguir voltar ao código inicial. Este trabalho dará preferência a desfazer primeiro as mutações mais complexas, que gerem mais instruções, para só depois tentar as mais simples, pois um código resultante de uma mutação mais complexa tem menos chance de ser sido produzido incidentalmente, sem participação da *engine* metamórfica. Como exemplo, a Tabela 3.1 apresenta transformações, que são partes de uma *engine* metamórfica. A reversão do metamorfismo exposto em (a), pode ser inviabilizada se a reversão do metamorfismo exposto em (b) for executado antes:

Tabela 3.1 – Exemplo de metamorfismos

Linha	**Código Metamorfisado**	**Código Original**
1	push eax	movsb
2	mov al, [esi]	
3	add esi, 1	
4	mov [edi], al	
5	add edi, 1	
6	pop eax	

(a) Metamorfismo complexo

linha	**Código Metamorfisado**	**Código Original**
1	mov al, [esi]	lodsb
2	add esi, 1	

(b) Metamorfismo simples

O código metamorfisado em (b) corresponde a um subconjunto deste código em (a), contido na segunda e terceira linha. Se estas linhas forem substituídas pelo código original em (b), ou seja, estas duas linhas substituídas pela instrução *lodsb*, o código original *movsb* pode não ser reconhecido.

Para os casos em que o código possa ter sido gerado por metamorfismo será aplicado o "metamorfismo reverso", de forma a garantir que todos os códigos originados por um mesmo

código metamórfico, após o processo de normalização, se apresentem semelhantes. Não importa se o código foi realmente resultado de um metamorfismo ou não, visto que o objetivo não é obter o código original, mas reconhecer códigos que possam ser o mesmo com diferentes estados de metamorfismo.

O algoritmo de normalização dos *tokens* será aplicado individualmente em cada um dos *tokens*. Existem várias operações independentes de mutação que serão desfeitas. Essas operações serão denominadas Op_1, Op_2,...Op_n. As operações consideradas mais complexas, que resultam em mais instruções, terão números menores (Op_1 será a mais complexa). O programa buscará no *token* para cada uma dessas operações a seqüência de instruções que seria resultante da mutação correspondente a esta operação. Quando acha desfaz a mutação e volta ao início do processo neste *token*. A Figura 3.5 descreve a operação.

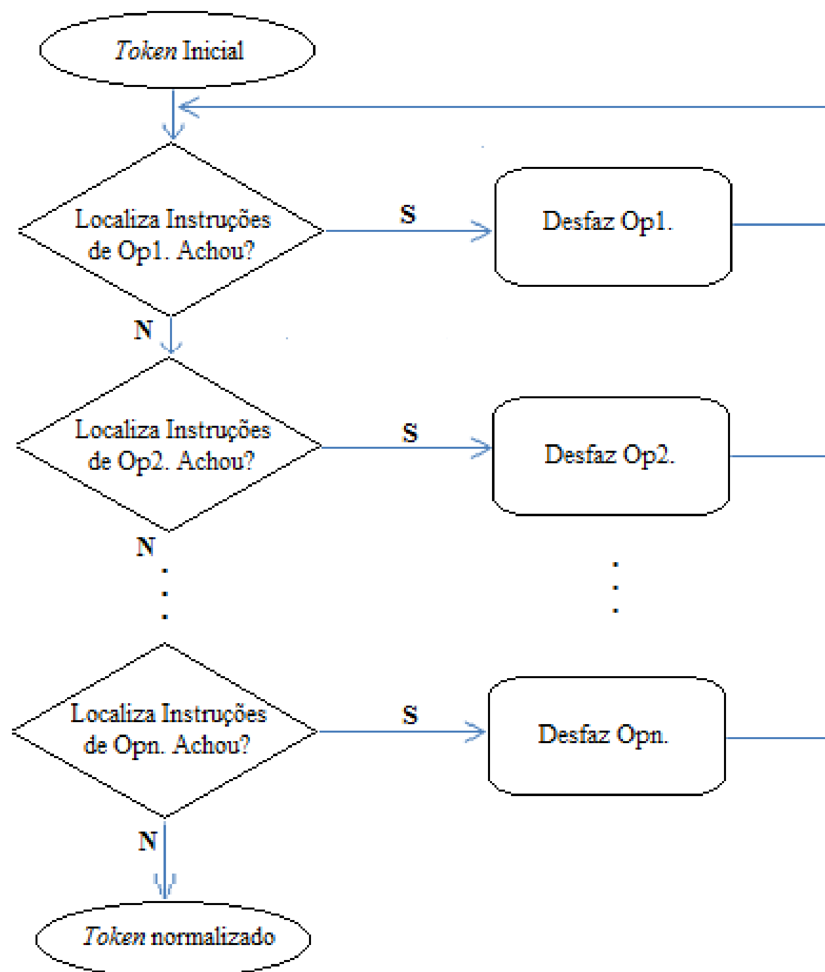

Figura 3.5 – Diagrama da normalização dos *tokens*

3.1.5. Cálculo de Impressões Digitais dos *tokens* (Etapa 5)

Impressão Digital do *token* (ID) é realiza através do cálculo de uma função hash utilizando o conteúdo considerado relevante do código *assembler* do *token*. Através dos caracteres ASCII das instruções e parâmetros considerados relevantes, nas linhas *assembler* correspondentes ao *token* é executado uma série de operações matemáticas de forma a resultar em um número de 32 bits usado como identificador. O algoritmo destas operações matemáticas foi desenvolvido pelo próprio autor deste trabalho. Neste cálculo de *hash* são consideradas relevantes as instruções e os parâmetros que não sejam endereços de desvio ou de endereço de chamadas de subrotinas. Essa diferença de tratamento nos endereços visa garantir que códigos equivalentes sejam reconhecidos como tal, independente do fato do endereço de desvio dos mesmos. Isto acontecerá quase sempre em códigos que sofreram metamorfismo, pois o código sofrerá reposicionamento devido as instruções inseridas, mudando os endereços de destino dos desvios.

O conjunto das IDs do *token* corrente e dos dois possíveis *tokens* seguintes na ordem de execução é chamado de Impressão Digital Composta (IDC). Este valor será usado no algoritmo de comparação.

3.2. PROCEDIMENTO DE DETECÇÃO

O procedimento de detecção consiste em repetir as etapas descritas na Seção 3.1 para cada um dos arquivos questionados e efetuar a comparação entre os dois resultados gerados. O processo de comparação é descrito nas próximas seções.

3.2.1. Comparação dos Códigos Normalizados

Nesta etapa ocorre a comparação dos IDC dos *tokens* dos arquivos questionados com os do *malware* buscado. Para cada coincidência na comparação será somado um ao somatório da comparação heurística. Essa etapa indica se o *malware* retratado na base de referência está presente no arquivo questionado. A Figura 3.6 apresenta o diagrama do processo de tratamento e comparação do arquivo questionado.

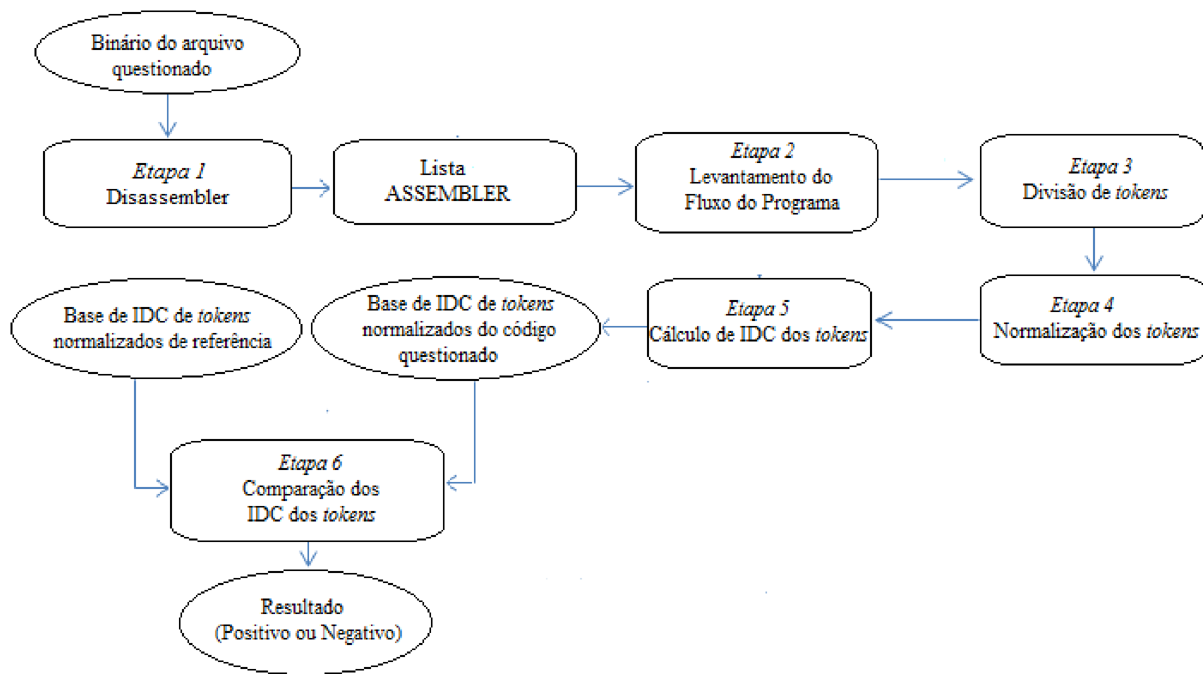

Figura 3.6 – Diagrama da detecção

Para cada *token* normalizado será comparado o IDC com o IDC de todos os *tokens* do *malware* buscado. Para cada coincidência incrementará um contador. O valor final do contador será usado para decidir se é ou não variante metamórfica.

3.2.2. Tratamento de Falsos Positivos

O resultado da comparação dos IDC pode dar resultado positivo devido à presença do *malware* buscado, mas pode também ocorrer por mera coincidência de trechos de código. Isso levaria a falsas identificações do *malware*. Quanto maior for o tamanho do arquivo sob teste maior será a probabilidade para ocorrência de coincidências indesejáveis. Assim, um programa grande, livre de contaminação, poderia somar uma pontuação heurística superior a um programa menor contaminado (com o *malware*). Para tratar este problema, um simples procedimento reconhecerá a ocorrência de falsos positivos, como segue: realiza-se novamente todo o procedimento anteriormente descrito, tanto para o *malware* quanto para o arquivo questionado, mas pulando a etapa de normalização. Se o resultado obtido for substancialmente menor que o obtido anteriormente, estará comprovada a presença do

malware. Quando não houver diferença substancial ou o *malware* não está presente ou está presente em sua versão de código viral original, sem metamorfismos.

3.3. CONSIDERAÇÕES FINAIS

Neste capítulo foi feita a descrição detalhada de todas as etapas da metodologia proposta. A avaliação sobre a presença de um *malware* é feita comparando-se o programa questionado com o *malware* buscado. A comparação é feita dividindo-se o programa a ser testado em pequenos grupos de instruções, chamados de *tokens*. É levantado também, para cada *token*, as possíveis instruções subsequentes durante a execução do programa. Faz se a busca para cada *token* do programa questionado, por um *token* no programa do *malware* buscado que tenha equivalência não só do *token* em questão como também nos possíveis *tokens* imediatamente subsequentes. O resultado total da comparação fornecerá um valor de pontuação heurística que, passando de determinado limiar, será considerada a presença do *malware*. Para reconhecimento de falsos positivos será repetido o processo, agora sem desfazer as mutações metamórficas. Não havendo diferença significativa entre a pontuação heurística das duas comparações, desfazendo e não desfazendo as mutações, será considerado que o *malware* não está presente.

4. EXPERIMENTAÇÕES PRÁTICAS

Este capítulo descreve um estudo de caso onde a metodologia proposta será aplicada no processo de verificação da presença de um *malware* metamórfico em um conjunto de arquivos suspeitos de contaminação. O capítulo está dividido em duas etapas: explicação das soluções implementadas para tratar o caso em análise, e apresentação dos resultados obtidos durante o experimento.

4.1. AMBIENTE DE EXPERIMENTAÇÃO

Todos os experimentos foram executados em um computador HP Pavilion dv6700 Notebook PC, com processador Intel Core2 Duo 2 GHz, 3 GB de memória RAM e com sistema operacional Windows Vista Home Premium SP1 (32 bits).

4.2. ESTUDO DE CASO

Neste estudo em particular, foi escolhido o vírus W32 Evol (Symantec, 2007), devido ao fato de possuir uma *engine* metamórfica que contempla as técnicas mais comuns empregadas para efeito de ofuscação de código sem, entretanto, fazer uso de cifragem, ou outras técnicas mais complexas de ofuscação que fugiriam ao escopo deste trabalho.

A ferramenta para realizar o processo de *disassembler* escolhida foi o Ida Pro Freeware versão 5.0 (Hex-Rays, 2011). Esta ferramenta foi escolhida por ser bastante reconhecida e por apresentar um bom desempenho. O Ida Pro é capaz de gerar listagens *assembler* legíveis e fáceis de trabalhar. Através desse software, códigos do *malware* serão desassemblados e será gerada uma listagem *assembler* do código já montado. A Figura 4.1 traz uma imagem do código desassemblado aberto na interface de edição do Ida Pro e a Figura 4.2 traz um trecho da listagem (resultado do processo de desassemblagem) deste mesmo código gerada pelo Ida Pro.

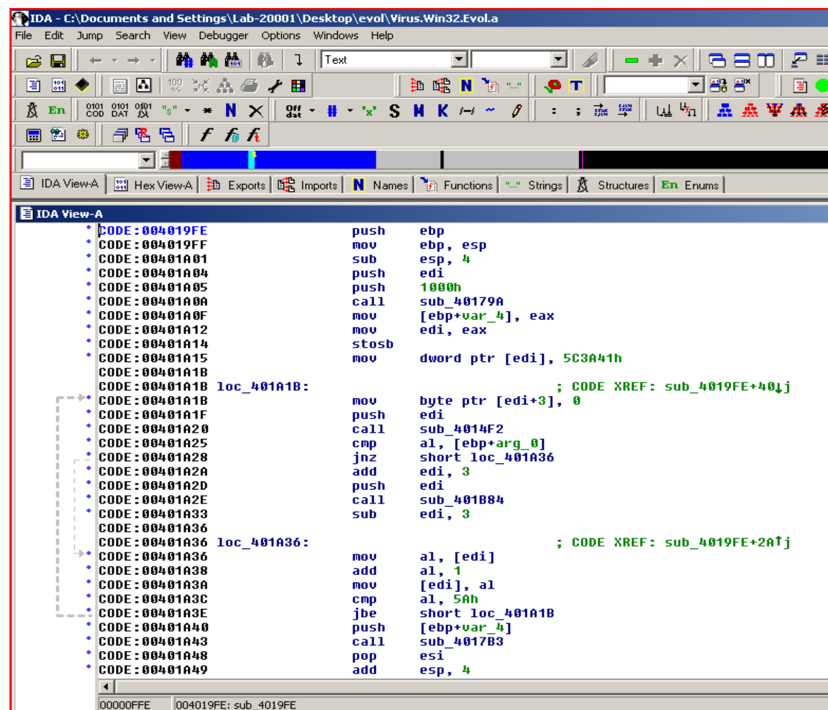

Figura 4.1 – Código do *malware* “W32 Evol a” aberto no Ida Pro

```
CODE:004019FE push    ebp
CODE:004019FF mov     ebp, esp
CODE:00401A01 sub     esp, 4
CODE:00401A04 push    edi
CODE:00401A05 push    1000h
CODE:00401A0A call    sub_40179A
CODE:00401A0F mov     [ebp+var_4], eax
CODE:00401A12 mov     edi, eax
CODE:00401A14 stosb
CODE:00401A15 mov     dword ptr [edi], 5C3A41h
CODE:00401A1B
CODE:00401A1B loc_401A1B:
CODE:00401A1B mov     byte ptr [edi+3], 0
CODE:00401A1F push    edi
CODE:00401A20 call    sub_4014F2
CODE:00401A25 cmp     al, [ebp+arg_0]
CODE:00401A28 jnz     short loc_401A36
CODE:00401A2A add     edi, 3
CODE:00401A2D push    edi
CODE:00401A2E call    sub_401B84
CODE:00401A33 sub     edi, 3
CODE:00401A36
CODE:00401A36 loc_401A36:
CODE:00401A36 mov     al, [edi]
CODE:00401A38 add     al, 1
CODE:00401A3A mov     [edi], al
CODE:00401A3C cmp     al, 5Ah
CODE:00401A3E jbe     short loc_401A1B
CODE:00401A40 push    [ebp+var_4]
CODE:00401A43 call    sub_4017B3
CODE:00401A48 pop     esi
CODE:00401A49 add     esp, 4
```

Figura 4.2 – Exemplo de listagem gerada pelo Ida Pro

4.2.1. Normalização

O processo de normalização inicia com o levantamento das mutações feitas pela *engine* metamórfica do *malware* W32 Evol. São selecionadas quais influenciarão no processo de comparação de *tokens* e são desfeitas, uma a uma, as mutações realizadas por esta *engine*. A seguir uma descrição das mutações presentes nesta *engine*.

Existem dois tipos principais de mutações:

1. Substituição de instruções por equivalentes - mutações que transformam uma instrução em várias instruções cujo resultado final seja o mesmo;
2. Inserção de código lixo – Inserção de código inerte, que não interfira no desenrolar do programa;

Para executar as contra-mutações (processo de reversão das mutações) no caso da inserção de código lixo e, ainda assim, manter o código funcional, seria necessário uma análise mais complexa do código para garantir que as instruções são inúteis de fato. Mas como o objetivo deste trabalho é encontrar as mutações de um código, e não mantê-lo funcionando, sempre que for possível que uma linha de código seja lixo proveniente de uma mutação ela será removida. Como a instrução será removida, tanto no código usado como padrão de comparação quanto no que está sendo verificado, a comparação não será afetada por uma remoção equivocada de suposto código lixo. As contra-mutações que consideram as mutações de substituição de instruções por equivalentes serão feitas através da comparação das partes dos *tokens* com os códigos alterados (aquele que sofreu mutação) contidos em uma tabela de mutações, descrita na Tabela 4.1.

As contra-mutações obedecerão a uma ordem de execução em cada *token.* As mudanças mais complexas, que gerem mais linhas de código, serão priorizadas durante o processo de reversão, visto um código resultante de uma mutação mais complexa tem menos chance de ser sido produzido incidentalmente, sem participação da *engine* metamórfica. Todas as contra-mutações, sempre que bem sucedidas, retornarão o processo ao início, reiniciando pelas contra-mutações de maior prioridade. As Tabelas 4.1 e 4.2 descrevem as mutações que são desfeitas.

Tabela 4.1 – Sequência de instruções normalizadas por substituição

Código Metamorfizado	Código Normalizado
mov al, [esi] add esi, 1	lodsb
mov eax, [esi] add esi, 4	lodsd

push eax
mov al, [esi]
add esi, 1
mov [edi], al
add edi, 1
pop eax
	movsb

push eax
mov [eax], esi
add esi, 4
mov [edi], eax
add edi, 4
pop eax
	movsd

mov edi, [al] add edi, 1	stosb
mov edi, [eax] add edi, 4	stosd
push reg1 mov reg1, reg2 op1 reg3/mem, reg1 pop reg1	op1 reg3/mem, reg2 op1 ϵ {adc, add, and, cmp, mov, or, sbb, sub, test, xor}

push reg1
mov reg1, reg2
op1 reg2, reg3/mem
mov reg2, reg1
pop reg1
	op1 reg2, reg3/mem
	
	op1 ϵ {adc, add, and, cmp, mov, or, sbb, sub, lea, xor}

push reg1 mov reg(8)1, imm8 op1 reg(8)2/mem8, reg(8)1 pop reg1	op1 reg(8)2/mem8, imm8 op1 ϵ {adc, add, and, cmp, mov, or, sbb, sub, test, xor}
nop	removido

Sempre que houver uma das instruções *push eax*, *push ecx* ou *push edx* pode ser que as instruções subsequentes sejam lixo inserido pela *engine* metamórfica. Nesse caso, obrigatoriamente a instrução afetará o mesmo registro que foi salvo na pilha (pelo *push*). Esses casos estão exemplificados na Tabela 4.2.

Tabela 4.2 – Instruções normalizadas por remoção de lixo

Código Metamorfizado	Código Normalizado
push reg32 mov reg32, [ebp+Random8]	push reg32
push reg32 op1 reg32, Random32 op1 ϵ {adc, add, and, mov, or, sbb, sub, xor}	push reg32
push reg32 mov reg8, Random8 reg8 ϵ reg32	push reg32

A *engine* metamórfica insere esse lixo esperando que imediatamente após o registro ter sido salvo na pilha o valor contido nele não será mais utilizado. Ao escrever o código do *malware* que sofrerá o metamorfismo, o programador tem que tomar o cuidado para que essa condição seja verdadeira ou o programa correrá o risco de não funcionar após o metamorfismo.

Existem ainda outras operações da *engine* metamórfica que não afetam a comparação dos códigos, e por isso não é necessário realizar o tratamento de normalização.

4.3. RESULTADOS

Esta seção apresenta os resultados de avaliação de um conjunto de arquivos que correspondem tanto a mutações do *malware* W32 Evol quanto a arquivos livres de contaminação. Tais arquivos foram submetidos a um conjunto de ferramentas (antivírus)

comerciais e à metodologia proposta.

Para produção de amostras de teste, a ação da *engine* metamórfica do W32 Evol foi emulada através de um programa em linguagem "C++", construído através da conversão direta do código em *assembler* desta *engine* (encontrado no código fonte do W32 Evol), para seu equivalente em linguagem "C++", que executa o processo de ofuscação de código em qualquer programa desejado. Esta conversão foi feita identificando a *engine* metamórfica do "W32 Evol a" em seu código binário através de um processo de engenharia reversa. Após isso o código da *engine* foi reproduzido linha a linha no programa em "C++" fazendo-se uso de inserção direta de código *assembler inline* sempre que possível.

O programa de metamorfismo foi executado uma ou mais vezes produzindo 63 versões do *malware* "W32 Evol.a". Cada execução da *engine* metamórfica sob um arquivo será chamada de "geração". Assim, dessas 63 versões, 32 foram feitas executando-se o programa de metamorfismo uma única vez (amostras de primeira geração), 16 duas vezes (amostras de segunda geração), 8 três vezes (amostras de terceira geração), 4 quatro vezes (amostras de quarta geração), 2 cinco vezes (amostras de quinta geração) e 1 seis vezes (amostra de sexta geração). Os experimentos apresentados nesta Seção mostram que as quantidades de amostras utilizadas são suficientes para apontar tendências comportamentais do *malware* avaliado.

4.3.1. Avaliação comparativa de antivírus comerciais

Para avaliar a identificação dos *malware,* usados no estudo de caso, em antivírus conceituados foram executados testes na ferramenta VirusTotal[1], que avalia arquivos suspeitos em 43 antivírus. Primeiramente foi testado o *malware* "W32 Evol.a", que foi identificado por 39 dos 43 antivírus. O resultado completo deste experimento está apresentado na Tabela 4.3.

Tabela 4.3 – Resultado do teste do *malware* "W32 Evol. a" no VirusTotal I

Antivirus	Malware Identificado
AhnLab-V3	Win32/Evol
AntiVir	W32/Evol.A
Antiy-AVL	Virus/Win32.Evol
Avast	Win32:Evol
AVG	Win32/Evol
BitDefender	Win32.Evol.A

[1] VirusTotal website, http://www.virustotal.com

ByteHero	Trojan.Win32.Heur.Gen
CAT-QuickHeal	W32.Evol.A
ClamAV	W32.Evol.a
Commtouch	W32/Evol.A
Comodo	Virus.Win32.Evol.a
DrWeb	Win32.Evol
Emsisoft	Virus.Win32.Evol.a!IK
eSafe	Win32.Evol.a
eTrust-Vet	Win32/Evol
F-Prot	W32/Evol.A
F-Secure	Win32.Evol.A
Fortinet	W32/Evol.A
GData	Win32.Evol.A
Ikarus	Virus.Win32.Evol.a
Jiangmin	Win32/Evol.a
K7AntiVirus	Virus
Kaspersky	Virus.Win32.Evol.a
McAfee	W32/Evol.dr
McAfee-GW-Edition	W32/Evol.dr
Microsoft	Virus:Win32/Evol.A.dr
NOD32	Win32/Evol.A.Gener1
Norman	W32/Evol.C
nProtect	Win32.Evol.A
Panda	Univ.B
PCTools	Malware.Evol
Prevx	-
Rising	Trojan.Win32.Generic.122E1CAE
Sophos	W32/Evol-A
SUPERAntiSpyware	-
Symantec	W32.Evol.Gen
TheHacker	-
TrendMicro	PE_EVOL_A.DR
TrendMicro-HouseCall	PE_EVOL_A.DR
VBA32	Malware-Cryptor.Win32.General.4
VIPRE	Trojan.Win32.Generic!BT
ViRobot	-
VirusBuster	Win32.DR.Evol.A

A Tabela 4.4 apresenta os resultados dos experimentos realizados no VirusTotal com 10 amostras, sendo 5 de primeira geração e 5 de segunda geração. São apresentados apenas os antivírus que obtiveram resultados de identificação positiva.

Tabela 4.4 – Resultado do teste do *malware* “W32 Evol a” no VirusTotal II

Amostra	**Geração**	**Antivirus**	**Malware Identificado**
1	1ª	AntiVir McAfee-GW-Edition PCTools Symantec	HEUR/Malware Heuristic.LooksLike.Win32.Suspicious.J Malware.Evol W32.Evol.Gen
2	1ª	AntiVir AVG McAfee-GW-Edition PCTools Symantec	HEUR/Malware Win32/Heur Heuristic.LooksLike.Win32.Suspicious.J Malware.Evol W32.Evol.Gen
3	1ª	AntiVir McAfee-GW-Edition	HEUR/Malware Heuristic.LooksLike.Win32.Suspicious.J
4	1ª	AntiVir AVG McAfee-GW-Edition PCTools Symantec	HEUR/Malware Win32/Heur Heuristic.LooksLike.Win32.Suspicious.J Malware.Evol W32.Evol.Gen
5	1ª	AntiVir McAfee-GW-Edition	HEUR/Malware Heuristic.LooksLike.Win32.Suspicious.J
6	2ª	McAfee-GW-Edition	Heuristic.LooksLike.Win32.Suspicious.J
7	2ª	Comodo McAfee-GW-Edition	Heur.Packed.Unknown Heuristic.LooksLike.Win32.Suspicious.J
8	2ª	McAfee-GW-Edition	Heuristic.LooksLike.Win32.Suspicious.J
9	2ª	McAfee-GW-Edition NOD 32	Heuristic.LooksLike.Win32.Suspicious.J a variant of Win32/Kryptik.AT
10	2ª	McAfee-GW-Edition	Heuristic.LooksLike.Win32.Suspicious.J

Os resultados dos experimentos no VirusTotal das variantes metamórficas foram insatisfatórios do ponto de vista de eficiência na detecção. Um máximo de 5 antivírus identificaram simultaneamente os arquivos submetidos, dos quais apenas 2 antivírus (PCTools e Symantec) identificaram em alguma amostra como sendo o *malware* Evol W32.

4.3.2. Avaliação usando a metodologia proposta

As 63 amostras metamórficas geradas foram comparadas com o "W32 Evol.a" usando a metodologia deste trabalho. Para cada arquivo foram feitas duas comparações: uma sem desfazer os metamorfismos e outra desfazendo. Os resultados, em pontuação heurística, destas comparações estão apresentados na Tabela 4.5. Além destas amostras foram também comparados 8 programas executáveis que não possuem o *malware* e três versões do *malware* W32 Evol – "W32 Evol a", "W32 Evol b" e "W32 Evol c" – disponíveis no site www.vxheavens.com. A metodologia deste trabalho foi aplicada uma vez para cada um destes arquivos. Os resultados destas comparações estão nas Tabelas 4.6 e 4.7. Este trabalho optou por utilizar o valor absoluto da pontuação heurística ao invés de fazer a normalização de valores, devido à falta de clareza de qual seria o procedimento mais adequado. Desta forma, o valor absoluto só fará sentido quando comparado com outros valores absolutos de um mesmo *malware*. O número de *tokens* do *malware* buscado e do arquivo questionado podem vir a serem fatores relevantes em critérios para normalização da pontuação heurística. A determinação do melhor método de normalização desta pontuação, bem como o melhor critério para determinar um valor (ponto de corte) que identifique a presença do *malware* no arquivo questionado poderão ser realizados no futuro, pois necessitam de uma quantidade e variedade de experimentos muito superior aos executados neste trabalho. Como será mostrado a seguir, os experimentos deste trabalho demonstram a viabilidade da metodologia e tendências de comportamento usando a pontuação heurística proposta (valor absoluto).

Tabela 4.5. Pontuação heurística da comparação do "W32 Evol.a" com suas versões metamórficas, separadas por geração de metamorfismo, desfazendo (D) e não desfazendo (ND) o metamorfismo.

Número da Amostra	Geração											
	1ª (ND)	1ª (D)	2ª (ND)	2ª (D)	3ª (ND)	3ª (D)	4ª (ND)	4ª (D)	5ª (ND)	5ª (D)	6ª (ND)	6ª (D)
1	40	126	3	65	3	47	2	26	2	10	2	9
2	36	117	2	63	2	40	2	31	0	15		
3	40	130	1	60	1	36	0	18				
4	39	129	1	59	0	35	0	21				
5	42	135	2	58	0	45						
6	35	117	1	65	2	49						
7	37	123	2	60	2	48						
8	36	124	1	51	2	43						

9	37	122	2	68								
10	26	116	4	67								
11	34	118	3	55								
12	42	127	3	56								
13	42	125	3	66								
14	36	125	2	62								
15	37	120	4	66								
16	23	123	4	64								
17	28	108										
18	42	125										
19	36	128										
20	36	121										
21	39	117										
22	39	120										
23	31	121										
24	37	124										
25	35	118										
26	38	129										
27	40	119										
28	42	131										
29	36	118										
30	38	118										
31	32	123										
32	40	130										
Média da pontuação heurística	36,59	122,72	2,38	61,56	1,5	42,88	1	24	1	12,5	2	9

Tabela 4.6. Pontuação heurística de similaridade do "W32 Evol.a" com arquivos livres de contaminação (sem o *malware*), desfazendo (D) e não desfazendo (ND) o metamorfismo.

Arquivo	**(ND)**	**(D)**	**Tamanho**	**Auto-comparação**
cacls.exe	0	0	25 KB	890
dialer.exe	0	0	31 KB	717
fveupdate.exe	0	0	13 KB	185
hh.exe	0	0	15 KB	140
Notepad.exe	1	1	148 KB	1119
Winhelp.exe	0	0	251 KB	500
winhlp32.exe	0	0	9 KB	54
wzsepe32.exe	0	0	204 KB	1388

Na última coluna está o resultado da comparação de cada arquivo com ele mesmo (auto-comparação), para conferência de funcionamento. A diferença de pontuação heurística entre os arquivos com o *malware* (veja a Tabela 4.5) e os sem *malware* (Tabela 4.6) foi tão grande que não foi necessário determinar um limiar mínimo que indique que o arquivo investigado está contaminado.

No universo de amostras testadas a diferença mínima na pontuação heurística desfazendo-se o metamorfismo para não desfazendo ocorreu na amostra de sexta geração e foi igual a 7(9-2). Foi observada uma tendência a diminuição desta diferença em cada geração do *malware* possivelmente devido a um aumento de falhas ao desfazer os metamorfismos quando em gerações de maior índice. Nas amostras sem o *malware* esta diferença foi igual a zero em todos os casos.

Todos os arquivos usados para avaliação de falsos positivos (arquivos não contaminados) foram menores que 1 MB de tamanho. Arquivos muito maiores, com muitas dezenas de MB, podem ter pontuação heurística acidental que deixe margem à dúvida da presença do *malware*. Neste caso pode ser usado não só a pontuação como também a diferença da pontuação na avaliação que desfaz os metamorfismos para a que não desfaz. Quando o *malware* está presente, em qualquer geração que não a do código viral original, esta diferença será grande.

Para comprovar que no caso de estar presente na condição do código viral original esta diferença não ocorre, foi realizado um experimento usando outras versões do "W32 Evol" com o código livre de qualquer tipo metamorfismo. A Tabela 4.7 apresenta os resultados obtidos.

Tabela 4.7. Pontuação heurística de similaridade do "W32 Evol.a" com outras versões do "W32 Evol", desfazendo (D) e não desfazendo (ND) o metamorfismo

Arquivo	(ND)	(D)	Tamanho	Auto-comparação
W32 Evol a	297	297	12 KB	297
W32 Evol b	293	293	12 KB	298
W32 Evol c	291	291	12 KB	298

Como observado na Tabela 4.7, nos resultados obtidos com a comparação do Evol.a com as outras três versões do Evol (obtidas no VXHeavens) não fez diferença desfazer ou não os

metamorfismos, o que caracteriza que eles não são versões metamórficas do mesmo *malware*, mas sim versões originais (sem metamorfismo) de programas ligeiramente diferentes entre si, o que foi confirmado posteriormente por observação direta dos três códigos..

4.3.3. Análise dos resultados

Para melhor entendimento dos resultados obtidos durante o processo de experimentação, e apresentados nas Tabelas 4.5 e 4.6, destacam-se as seguintes observações:

1. Cada ponto na comparação heurística pode ocorrer por duas razões: pela presença do *malware* ou, por coincidências incidentais dos *tokens*. Coincidências podem ser provenientes do arquivo questionado usar estruturas semelhantes as existentes no *malware* ou por colisão no algoritmo de *hash* do cálculo na impressão digital composta;
2. Um *malware* comumente estará em uma de três situações:
    i. Acrescido ao código do programa hospedeiro tornando-o maior. Neste caso, a pontuação heurística do arquivo questionado será provavelmente igual a soma da pontuação do *malware* isolado mais a pontuação incidental do programa hospedeiro original (sem o *malware*);
    ii. Substituindo uma parte inerte (sem código ou dados) do programa original, para dificultar a detecção do *malware* pelo aumento do tamanho do executável original. Neste caso, o *malware* deve estar sobrepondo uma região onde não há código e, portanto, é muito improvável que haja uma coincidência incidental (que estaria sendo destruído pelo *malware*). Assim a provável pontuação heurística do arquivo questionado será também a soma da pontuação do *malware* isolado mais a pontuação incidental do arquivo hospedeiro original (sem o *malware*);
    iii. Não possuir programa hospedeiro, estando situado em arquivo só com o *malware*;

3. Não foram feitos testes nas duas primeiras condições (item 2, i e ii), com arquivo hospedeiro infectado, tendo sido feito unicamente com o *malware* isolado, ficando o teste nas outras condições como sugestão para trabalhos futuros;
4. A metodologia foi testada somente em 8 arquivos sem o *malware* (livre de contaminação), todos de tamanho igual ou inferior a 251KB. Neste universo ocorreu em apenas um caso com 1 (um) ponto heurístico por coincidência incidental;
5. Dentre os arquivos com *malware* testados mostrados na Tabela 4.5 foram levantadas as pontuações heurísticas máxima e mínima conseguidas dentre as amostras de cada geração desfazendo o metamorfismo, para cada geração do *malware*. Tal critério pode ser usado para a identificação do *malware*, conforme mostra a Tabela 4.8
6. Como complemento ao critério de identificação do *malware* também foram empregadas as diferenças máxima e mínima desfazendo e não desfazendo o metamorfismo, para cada geração do *malware*. Estes valores também são apresentados na Tabela 4.8.

Tabela 4.8 . Diferenças máxima e mínima entre a pontuação heurística desfazendo (D) e não desfazendo (ND) o metamorfismo e pontuação heurística máxima e mínima para cada geração do arquivos de *malware*.

Geração	Diferença mínima entre (D) e (ND)	Diferença máxima entre (D) e (ND)	Pontuação mínima (D)	Pontuação máxima (D)
1	78	100	116	131
2	50	66	51	68
3	35	47	35	49
4	18	29	18	31
5	8	15	10	15
6	7	7	9	9

7. Se as amostras apresentadas na Tabela 4.6 não estivessem sozinhas, e sim anexadas em algum arquivo hospedeiro (casos 2.i e 2.ii), a diferença de pontuação permaneceria a mesma. Isto ocorreria porque as pontuações extras decorrente de identificação de estruturas similares (incidentais) ao vírus estariam presentes nos dois

momentos do arquivo questionado (normalizado e não normalizado). Isto seria verdade com os 8 arquivos livre de contaminação com os quais foram feitos os testes (veja tabela 4.6), mas nem sempre esta diferença será nula, sendo sugerida a avaliação deste comportamento para arquivos maiores em trabalhos futuros.

8. Nos casos testados, não houve interseção entre as regiões de máximo e mínimo nem de pontuação e nem de diferença entre as diferentes gerações, sugerindo ser possível identificar inclusive a geração do *malware* presente no arquivo pela pontuação heurística final, dependendo do intervalo. No entanto, principalmente nas gerações mais elevadas, as amostradas utilizadas foram pequenas para tal afirmação, necessitando de uma avaliação mais ampla para uma conclusão categórica.
9. Analisando os resultados obtidos, para arquivos questionados livres de contaminação, de até 251KB de tamanho, os resultados sugerem que um valor de pontuação heurística entre 1 e 9 ou uma diferença de pontuação, desfazendo e não desfazendo o metamorfismo, entre 0 e 7, poderiam ser usados como limiar para identificação deste *malware*. Isto foi verdade para o caso particular deste *malware* com os arquivos usados nos testes. Para determinar um algoritmo que determine os valores deste limiar para um caso genérico serão necessários muito mais testes com amostras muito mais variadas.

4.4. RESUMO DO CAPÍ|TULO

Neste capítulo foram descritos os experimentos realizados. Foram geradas amostras de *malware* de várias gerações e testadas em antivírus comerciais e na metodologia proposta. Foram também testados arquivos sem contaminação para efeito de controle de falsos positivos. No final foi feita uma análise dos resultados.

5. CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho apresentou uma metodologia para identificação de variantes metamórficas de um *malware*. A metodologia se baseia em fazer uma identificação por "assinaturas funcionais" comuns ao *malware* e o arquivo questionado, buscando partes dos programas que tenham funções similares. Para fazer esta identificação os programas são antes passados por um processo de "normalização", no qual se tenta desfazer alterações metamórficas previstas em uma "biblioteca" de metamorfismos conhecidos.

Os resultados obtidos nos experimentos foram bastante promissores, tendo identificado 100% dos arquivos que continham o *malware* e 0% de falsos positivos com os arquivos testados sem *malware*. Durante os experimentos foram identificas que três versões diferentes do *malware* testado não eram versões de um mesmo *malware* original metamorfoseadas, mas sim versões ligeiramente diferentes do *malware*.

A cifragem e compactação de código não são tratados nesta metodologia. Como trabalho futuro pretende-se tratar *malware* que possuam tais características. Para isso será necessário usar a metodologia em conjunto com algum tratamento para estes casos. No caso de não ser possível acessar o código decifrado em análise estática esse tratamento poderia ser aplicado (usando a metodologia) de forma dinâmica como, por exemplo, usando o conteúdo da memória.

Outro trabalho interessante é analisar arquivos maiores, visto que eles podem gerar pontuações heurísticas altas por simples coincidência. Seria interessante o aprimoramento dos testes com arquivos sem o *malware* de dimensões maiores e, caso ocorram situações de falso positivo, buscar soluções para minimizá-las. Um algoritmo de normalização da pontuação heurística, relevando o tamanho, ou número de *tokens*, do arquivo questionado e do *malware* buscado, pode se fazer necessário.

A questão da ordem em que são desfeitas as alterações metamórficas pode ser aprimorada inclusive prevendo a possibilidade de testar em várias ordens diferentes, gerando mesmo várias IDC diferentes para um mesmo *token*.

Por fim, a ampliação da biblioteca de metamorfismos, contemplando metamorfismos de outras *engines* metamórficas, e aprimoramento de metodologias de desfazer metamorfismos também são aprimoramentos desejáveis.

REFERÊNCIAS

E. Skoudis. “*Malware: Fighting Malicious Code*.” Prentice-Hall, 2004

Morin, M “The Financial Impact of Attack Traffic on Broadband Networks“. *IEC Annual Review of Broadband Communications*, páginas 11-14, 2006.

“Annual Report PandaLabs 2009”, Disponível em: <http://www.pandasecurity.com/img/enc/Annual_Report_PandaLabs_2009.pdf>. Acesso em: 5 de Outubro de 2011.

Symantec – Internet Security Threat Report.Volume 16, 2010.
Disponível em: <http://www.symantec.com/business/threatreport/build.jsp>. Acesso em: 5 de Outubro de 2011.

Wing Wong and Mark Stamp. “Hunting for metamorphic engines.” Journal in Computer Virology, 2:211–229, 2006.

Szor Peter. “The Art of Computer Virus Research and Defense“ Addison Wesley Professional, 2005.

David, M. A.; Bodmer, Sean M.; LeMasters, A. “Hacking Exposed – Malware & Rootkits Secrets & Solutions”. The McGraw-Hill Companies, 2010.

Eilam, E. "Reversing - Secrets of Reversing Engineering". Wiley Publishing, Inc., Indianapolis, Indiana, *foreword* pag.VIII, 2005.

Stone-Gross, B.; Cova, M.; Cavallaro, L.; Gilbert, B.; Szydlowski, M.; Kemmerer, R.; Kruegel, C. and Giovanni V.; “Your Botnet is My Botnet: Analysis of a Botnet Takeover”, Department of Computer Science, University of California, Santa Barbara, 2009.
Disponível em <http://www.cs.ucsb.edu/~seclab/projects/torpig/torpig.pdf>. Acesso em: 07 de Outubro de 2011.

Spafford, Eugene H., “A Failure to Learn from the Past”, Purdue University CERIAS, 2003

Zelonis, K. “AVOIDING THE CYBER PANDEMIC: A Public Health Approach to Preventing Malware Propagation", 2003.

Info Security – “Mytob tops list of most significant virus over last 40 years” (18 March 2011). Disponível em: <http://www.infosecurity-magazine.com/view/16726/mytob-tops-list-of-most-significant-virus-over-last-40-years/>. Acesso em: 11 de Outubro de 2011.

InformationWeek – “Storm Worm Botnet More Powerful Than Top Supercomputers”, 2007. Disponível em: <http://www.informationweek.com/news/201804528>. Acesso em: 11 de Outubro de 2011.

G. *Lawton*, *On the Trail of the Conficker Worm*, in Technology News, IEEE, pp *19-22*, June 2009.

“Iran's Nuclear Agency Trying to Stop Computer Worm”. New York Times. 25-9-2010.

“Computer virus”.
Disponível em: <http://microsoft.wikia.com/wiki/Computer_virus>. Acesso em: 14 de Outubro de 2011.

Venkatesan, Ashwini. “CODE OBFUSCATION AND VIRUS DETECTION”, San Jose State University, May, 2008.

Handle – Boccardo, D. R.; Machado, R. C. S. C. ; Costa, L. F. R. da; “Transformações de código para proteção de software”, 2010.
Disponível em: <http://hdl.handle.net/123456789/397>. Acesso em: 3 de Novembro de 2011.

Cullen Linn, Saumya Debray; Department of Computer Science University of Arizona, “Obfuscation of Executable Code to Improve Resistance to Static Disassembly”,2003.

Zhang, Q., “Polymorphic and metamorphic malware detection", Ph.D. Thesis thesis, ^Graduate Faculty, North Carolina State University, Raleigh, NC, USA, 2008.

Andrew, Walenstein; Mathur, Rachit; Chouchane ,Mohamed R. e Lakhotia, Arun; “Normalizing Metamorphic Malware Using Term Rewriting”, Center for Advanced Computer Studies, University of Louisiana at Lafayette, 2006.

Eduardo Mazza Batista, “ASAT: uma ferramenta para detecção de novos vírus” - Universidade Federal de Pernambuco, 2008.

Matthew G. Schultz.; Eleazar E.; Erez Z.; Salvatore J. S. “Data mining methods for detection of new malicious executables”. In Proceedings of the 2005 IEEE Symposium on Security and Privacy, pag. 38–49. IEEE Computer Society, 2001.

Mohamed R. Chouchane and Arun Lakhotia. “*Using engine signature to detect metamorphic malware*”. In WORM ’06: Proceedings of the 4th ACM workshop on Recurring malcode, 2006.

Rieck, Konrad. Holz, Thorsten. Willems, Carsten. Düssel, Patrick. Laskov, Pavel. “Learning and Classification of Malware Behavior”. In Proceedings of Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), pages 108–125, 2008.

Mihai Christodorescu1, Somesh Jha1, Johannes Kinder2, Stefan Katzenbeisser2, and Helmut Veith2 1 University of Wisconsin, Madison, 2 Technische Universit¨at M¨unchen, “Software Transformations to Improve Malware Detection”,2007.

Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Code normalization for self-mutating malware. Security and Privacy, IEEE, 3(1):46–54, 2007.

K. Kim, and B. Moon, Malware Detection based on Dependency Graph using Hybrid Genetic Algorithm, Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation, pp. 1211-1218, 2010.

Symantec - W32. Evol, (February 13, 2007) Disponível em: <http://www.symantec.com/security_response/writeup.jsp?docid=2000-122010-0045-99>. Acesso em: 15 de Novembro de 2011.

Hex-Rays - IDA (October 13, 2011)
Disponível em: <http://www.hex-rays.com/products/ida/index.shtml>. Acesso em: 18 de Novembro de 2011.

OllyDbg 2.01 alpha 4 (03/08/2011) <http://www.ollydbg.de/>. Acesso em: 08 de Março de 2012.

SorceForge – Bastard (2011) <http://sourceforge.net/projects/bastard/>. Acesso em: 08 de Março de 2012.

Mario Schallner , 2004 – dispoonível em <http://lida.sourceforge.net/> . Acesso em: 08 de Março de 2012.