Universidade Federal do Amazonas
Instituto de Computação
Programa de Pós-Graduação em Informática

Identificação de *Malware* Metamórfico baseado em Grafos de Dependência

Gilbert Breves Martins

Manaus – Amazonas - Brasil
Março 2017

Universidade Federal do Amazonas
Instituto de Computação
Programa de Pós-Graduação em Informática

Identificação de *Malware* Metamórfico baseado em Grafos de Dependência

Tese submetida ao Programa de Pós-Graduação do Instituto de Computação da Universidade Federal do Amazonas para obter o título de Doutor em Informática.

Gilbert Breves Martins

Orientador: Prof. Eduardo James Pereira Souto, DSc.
Co-Orientadora: Prof.ª Rosiane de Freitas Rodrigues, DSc.

Manaus – Amazonas - Brasil
Março 2017

Ficha Catalográfica

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

Martins, Gilbert Breves
M386i Identificação de Malware Metamórfico baseado em Grafos de Dependência / Gilbert Breves Martins. 2017
88 f.: il. color; 31 cm.

Orientador: Eduardo James Pereira Souto
Coorientadora: Rosiane de Freitas Rodrigues
Tese (Doutorado em Informática) - Universidade Federal do Amazonas.

1. Malware. 2. Metamorfismo de Código. 3. Grafos de Dependência. 4. Segurança. I. Souto, Eduardo James Pereira II. Universidade Federal do Amazonas III. Título

FOLHA DE APROVAÇÃO

“Identificação de *Malware* Metamórfico baseado em Grafos de Dependência”

Gilbert Breves Martins

Manaus – Amazonas – Brasil
Março de 2017

Agradecimentos

Acima de tudo, agradeço ao bom Deus pelo dom da vida e por todas as oportunidades que sempre me são oferecidas. Agradeço aos meu pais, Gilberto Alves Martins e Maria Vitória Breves Martins, por serem minha referência de ética e idoneidade, por terem me ensinado que a educação é a maior ferramenta de evolução do homem, e por terem me mostrado que humildade e respeito ao próximo é uma chave que abre muitas portas na sua vida. Agradeço à minha esposa, Elisângela Leitão de Oliveira, e a meus filhos, Caio de Oliveira Martins e Diana de Oliveira Martins, por todo o apoio moral e paciência que demostraram durante todo o período de desenvolvimento desta pesquisa. Agradeço aos meus colegas de estudo da UFAM pelas horas compartilhadas em nossas atividades de estudo e trabalho. Sem eles, aqueles momentos teriam sido muito solitários. Finalmente, um agradecimento especial para meu orientador, Eduardo James Pereira Souto, e para minha Co-Orientadora, Rosiane de Freitas Rodrigues, pela enorme paciência, carinho, apoio e dedicação que demostraram durante todos os momentos que precisei de ajuda. Nada disto teria sido possível sem vocês.

Resumo

Comparar um programa com um conjunto de partes de código, conhecido como assinaturas, previamente armazenadas e extraídas de programas maliciosos previamente identificados, é a forma tradicional de se identificar se este programa também se trata ou está contaminado por um código malicioso. Para tornar este processo de identificação ineficaz, os desenvolvedores de programas maliciosos podem inserir em suas criações a capacidade de alterar a forma com o seu código se apresenta, mudando o corpo do código à medida que o processo de contaminação ocorre e invalidando o processo de identificação tradicional. Uma das maneiras de lidar com esta capacidade de mutação, também conhecido como metamorfismo de código, é baseada na geração de grafos que modelem as relações de dependência existentes entre os elementos do código, uma vez que estas relações se persistem, mesmo diante da mutação do código. Estes grafos são também conhecidos como grafos de dependência. Similar ao tradicional modelo de assinaturas, a identificação dos códigos maliciosos baseadas em grafos de dependência, ocorre quando o grafo gerado a partir do programa sob investigação e comparado com um conjunto de grafos previamente armazenados em uma base de referência, construída a partir da extração dos grafos de dependência de instâncias de códigos maliciosos previamente identificados. Como o processo de comparação entre grafos pertence à classe de problemas *NP-Difícil*, é necessário encontrar alternativas viáveis para tratar este problema, tornando a comparação entre grafos uma alternativa viável para a identificação de códigos maliciosos metamórficos. Usando grafos de dependência extraídos a partir de códigos executáveis, esta tese apresenta uma abordagem para reduzir o tamanho dos grafos de dependência usados no processo de comparação, pela introdução da diferenciação entre os vértices, com base nas relações de dependência características de cada deles possui. Combinada com a inclusão de arestas virtuais, esta metodologia possibilita a construção de um clique virtual que é utilizado para identificar e descartar as porções menos relevantes do grafo de dependência original, diminuindo o tamanho do grafo que será inserido na base de referência. Os resultados apresentados nesta tese também demonstram que esta redução aprimora o processo de identificação, diminuindo o coeficiente de variação dos resultados e aumentando a taxa de identificações de códigos maliciosos metamórficos.

Abstract

The traditional way to identify malicious programs is to compare the code body with a set of previous stored code patterns, also known as signatures, extracted from already identified *malware* code. To nullify this identification process, the *malware* developers can insert in their creations the ability to modify the *malware* code when the next contamination process takes place, using obfuscation techniques. One way to deal with this metamorphic *malware* behavior is the use of dependency graphs, generated by surveying dependency relationships among code elements, creating a model that is resilient to code mutations. Analog to the signature model, a matching procedure that compares these graphs with a reference graph database is used to identify a *malware* code. Since graph matching is a NP-hard problem, it is necessary to find ways to optimize this process, so this identification technique can be applied. Using dependency graphs extracted from binary code, we present an approach to reduce the size of the reference dependency graphs stored on the graph database, by introducing a node differentiation based on its features. This way, in conjunction with the insertion of virtual paths, it is possible to build a virtual clique used to identify and dispose of less relevant elements of the original graph. The use of dependency graph reduction and the node differentiation also produces more accurate results for the matching process. To validate these statements, we present a methodology for generating these graphs from binary programs and the results achieved with the use of all the proposed features for the identification of some metamorphic malware samples.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A intrusão hostil e indesejada de usuários ou softwares não autorizados, é um dos problemas mais significativos encontrados por aqueles que são responsáveis pelo controle da segurança em sistemas computacionais. O objetivo destes ataques é obter acesso a recursos disponíveis no sistema que, originalmente, estariam fora do alcance dos perpetradores do ato de invasão [1].

Programas maliciosos, ou *malware* (do inglês, *malicious software*), são uma das ferramentas mais comuns empregadas pelos atacantes para viabilizar atividades maliciosas. O termo *malware* 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)[2]. Vírus de computador, *worms*, *trojan horses* (cavalos de Tróia) e *spywares* são alguns exemplos de *malware.*

Para determinar se um arquivo é uma instância de um *malware*, as ferramentas desenvolvidas para este fim empregam tradicionalmente mecanismos que comparam pequenos trechos de códigos extraídos de programas maliciosos previamente identificados e armazenados em uma base de dados, com o corpo do arquivo suspeito [3]–[7]. Caso qualquer um destes trechos de código, também conhecidos como assinaturas, seja encontrado, o arquivo analisado é marcado como contaminado. Além disso, este processo também identifica a fonte de contaminação, baseado no *malware* de onde foi extraída a assinatura encontrada dentro do arquivo contaminado.

Entretanto, como este processo de comparação exige uma correspondência exata entre pelo menos um trecho de código do arquivo analisado e uma das assinaturas armazenadas na base de referência, alguns desenvolvedores de *malware* passaram a utilizar técnicas que tornam suas criações capazes de gerar versões diferentes de seu próprio código, durante o processo de contaminação do sistema alvo, de forma automática [8]. Como cada uma das novas versões é incompatível com qualquer

assinatura previamente coletada, o processo de identificação tradicional é completamente ludibriado.

Um exemplo deste tipo de técnica, apresentado por Yan e Wu [08], emprega programas que compactam e criptografam os arquivos executáveis que constituem o *malware* e que restauram a imagem executável original apenas no momento que o vírus é carregado para a memória da máquina infectada. Todavia, o programa pode novamente ser identificado por um processo de varredura tradicional que analise os códigos carregados na memória do dispositivo infectado.

Para fugir desta limitação, outra técnica tem sido empregada pelos desenvolvedores de códigos maliciosos para produzir diferentes versões do mesmo *malware* através de pequenas alterações na codificação original. Esta técnica, conhecida como *ofuscação de código* [9], se baseia na modificação da sequência original de instruções, alterando a forma como o código está escrito sem entretanto, gerar qualquer modificação na funcionalidade original dos trechos alterados. Essa capacidade de alteração do código também é conhecida como *metamorfismo* [10].

As versões metamórficas de um *malware* são geradas automaticamente por um componente do código que tem a função de executar as modificações no *malware* à medida que novas cópias são produzidas e propagadas. Exemplos de técnicas comuns de metamorfismo incluem [9]: *i)* a inserção de instruções e variáveis irrelevantes, também conhecida como inserção de código lixo, que não alteram a lógica original do programa; *ii)* a alteração no nome de variáveis ou troca mútua de variáveis entre instruções diferentes; *iii)* a substituição de sequências de instruções por outras que produzam o mesmo resultado; e *iv)* a alteração na ordem de execução das instruções, seja pelo reposicionamento de blocos de código independentes ou pelo uso de instruções de desvio de fluxo. O processo de metamorfismo pode ser obtido por programas externos, mas para que este processo seja totalmente automatizado, o próprio código malicioso deve ser capaz de gerar novas versões metamórficas de si mesmo, sem qualquer necessidade de intervenção humana [11].

Para lidar com os desafios da identificação de códigos maliciosos metamórficos, este trabalho propõe uma modelagem teórica que aprimora o uso de grafos de dependência como forma de identificação de códigos metamórficos executáveis. Esta modelagem teórica contribui tanto para a redução do tamanho das bases de referência como para o aprimoramento do processo de medição do nível de similaridade entre os

códigos comparados.

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. Nesse contexto, para combater tais ameaças, vários produtos comerciais e não comercias têm sido empregados como os anti-* (antivirus, *antimalware, antiphishing*), *firewalls*, sistemas de detecção de intrusão, entre outros. Entretanto, de acordo com Wong e Stamp [12], novos artifícios têm sido empregados pelos desenvolvedores de vírus para dificultar o processo de detecção, como a utilização de técnicas de criptografia, mutação do código viral, incluindo metamorfismo e polimorfismo, entre outras. Como o número de diferentes versões de um mesmo *malware* pode crescer exponencialmente, torna-se praticamente impossível sua detecção usando apenas assinaturas.

Algumas ferramentas comerciais têm privilegiado o uso de heurísticas no lugar de assinaturas. Entretanto, segundo Yin et al. [3], as heurísticas empregadas por estas ferramentas não são baseadas nas características fundamentais de um *malware*, o que pode levar a altas taxas de falsos negativos e falsos positivos. Além disso, cientes deste procedimento de detecção, novos *malware* podem ser desenvolvidos para evitar os elementos do sistema normalmente monitorados por este tipo de solução e, assim, dificultar o processo de detecção.

Diversos pesquisadores têm proposto abordagens alternativas para lidar com este problema. Alguns exemplos destas abordagens incluem: *i)* a criação de um padrão de assinatura capaz de identificar grupos de códigos através de uma única sequência de identificação [5]; *ii)* a utilização de autômatos finitos para modelar chamadas de sistemas associadas ao comportamento de códigos maliciosos [13]; *iii)* o levantamento das características comportamentais capazes de identificar uma família de *malware* [8]; *iv)* a normalização do código e o levantamento do fluxo para reversão e identificação de códigos suspeitos [14]; e *v)* a utilização de grafos para modelar o uso de funções [11] ou a relação de dependência entre instruções do código [15].

Estas propostas ainda não oferecem uma solução definitiva para o problema de identificação de códigos maliciosos metamórficos, visto que todas possuem algum tipo de limitação, como por exemplo: a criação manual dos modelos de detecção; a

quantidade de informações tratadas; a necessidade de carregamento do programa suspeito para a memória; ou ainda a alta variância nos resultados de identificação.

Especificamente, a utilização de grafos para modelar o comportamento dos *malware* tem sido vista como uma abordagem promissora, visto que esta versátil estrutura de dados tem a capacidade de modelar características presentes nos códigos que independem do seu modelo de codificação, gerando modelos mais resistentes às ações de metamorfismo de código. Entretanto, duas questões precisam ser superadas para seu uso no processo de identificação de códigos metamórficos. Primeiro, quando utilizados para modelar as relações de dependência entre as instruções do código, os grafos gerados possuem um conjunto de características semânticas que ainda não foram exploradas em todo o seu potencial, como observado nos trabalhos [11][15]. Além disso, algumas características estruturais dos grafos de dependência inviabilizam inicialmente a aplicação de algoritmos simples para a identificação das inter-relações existentes entre os seus componentes.

Assim, para lidar com essas duas questões e oferecer novas contribuições nesta área de pesquisa, este trabalho apresenta uma metodologia de classificação dos vértices que, em conjunto com a inclusão de estruturas virtuais dinâmicas, pretende servir de base tanto para a construção de um processo de identificação dos elementos estruturais mais relevantes, quanto para otimizar o processo de comparação entre os grafos de dependência analisados.

1.2. Justificativa

Durante a última década, tem havido um aumento significativo no número e na sofisticação dos ataques relacionados a ameaças digitais como vírus de computador, *botnets* e *worms* [16]. A forma de propagação destas ameaças também está se diversificando. Além dos tradicionais e-mails maliciosos e da propagação automática pela invasão de computadores pessoais conectados à Internet, Liu et al. [17] e Yagi et al. [18] identificam uma forma mais passiva de propagação, onde as próprias vítimas do ataque são induzidas a instalar aplicativos maliciosos. Isto é ilustrado pelo trabalho de Jeong [19], que apresenta um estudo de caso onde sites populares eram alvos de ataques que, por meio da exploração de falhas de segurança nos servidores web, tinham o objetivo de transformá-los em agentes propagadores de programas maliciosos. Tais programas eram usados para coletar informações pessoais dos usuários desses sites

como contas, senhas bancárias e dados de cartões de créditos, gerando um grande banco de dados que era depois vendido em troca de dinheiro e utilizado em fraudes financeiras.

Este tipo de ataque é definido como *Drive-By-Download* (DBD), onde são exploradas as vulnerabilidades dos navegadores (*browsers*) ou de outro elemento externo associado aos mesmos, através da ação do próprio usuário que ativa um link insuspeito e acaba por instalar um código malicioso em seu sistema [9] [20].

Outro aspecto preocupante é o aumento significativo no número de ataques a sistemas computacionais. Segundo Griffin et al. [5], a Symantec, empresa americana que trabalha na área de softwares antivírus, reportou que no ano de 2008, foram identificadas mais de um milhão de novos *malware*. Este número é maior do que a soma de todas as ocorrências dos anos anteriores. A empresa de segurança *Panda Labs*, em relatório a respeito das atividades de *malware* no anos de 2014 [21] afirma que, neste ano, foram coletadas mais de 220 milhões de novas amostras de *malware*, um número que corresponde à 34% de todos os *malware* já coletados até então. Tahan et al. [22] observam que no intervalo de tempo compreendido entre lançamento de novo vírus e a sua efetiva inclusão no banco de dados das ferramentas de detecção (por exemplo, antivírus), este novo vírus pode se propagar livremente.

Já há alguns anos, *softwares* maliciosos têm sido responsáveis por perdas financeiras que ultrapassam centenas de milhões de dólares anuais. Por exemplo, Morin [23] afirma que, em 2006, os prejuízos ocasionados por *worms* foram de aproximadamente US$ 245 milhões, somente entre provedores de acesso norte-americanos. Segundo o Relatório Anual da Panda Labs de 2009 [24], somente naquele ano surgiram mais novos códigos maliciosos do que em todos os anos anteriores. Nesse mesmo ano, o Pentágono gastou com recuperação de danos causados pelos *malware* aproximadamente US$ 100 milhões. Em 2014 a NSA (*National Security Agency*) afirmou que os custos relacionados a crimes cibernéticos ultrapassam o valor de US$ 385 milhões em nível mundial [25].

No contexto de América Latina (AL), o Brasil é o principal gerador de atividades maliciosas desde 2010, chegando a ter 44% de toda a atividade naquele ano [26], o que deixa o Brasil em uma posição relevante em nível mundial, tanto como vítima quanto como produtor. No que se refere à atividade de softwares maliciosos, as atividades maliciosas houve um aumento de 7% no primeiro trimestre de 2015 e, neste

mesmo período, a quantidade de amostras coletadas foi superior àquela encontrada em qualquer um dos anos anteriores [27].

Como exemplo desse crescimento, podemos citar o expressivo aumento dos ataques de negação de serviços (do inglês, *Distributed Denial of Service* - DDoS) sofridos pelos sites de órgãos governamentais em 2011 [28][29]. Este tipo de ataque é normalmente executado a partir de um grande conjunto de hosts previamente comprometidos por códigos maliciosos, onde o tráfego do ataque viaja através da Internet, em direção à rede de acesso da vítima.

Por tudo isto que foi exposto, fica clara a necessidade de busca de formas alternativas de identificação de códigos maliciosos que independam da decisão humana para a construção dos modelos de identificação que sejam capazes de representar mais de uma instância de *malware*.

1.3. Objetivos

Este trabalho tem como objetivo aprimorar o processo de identificação e classificação de códigos maliciosos metamórficos, através do uso de grafos de dependência para modelagem da correlação entre instruções e variáveis, por meio da introdução de um processo de classificação de vértices e do uso de estruturas virtuais dinâmicas. A abordagem proposta possibilita a criação de uma base referência constituída de grafos de dependência aprimorados, permitindo aprimoramento do processo de comparação entre o código executável de programas suspeitos e o código de *malware* conhecidos.

Para atingir o objetivo geral desta pesquisa, um conjunto de resultados intermediários foram estabelecidos:

1. Construção de um modelo teórico que permita a classificação dos diferentes tipos de vértice, o que possibilita a identificação dos elementos mais relevantes dos grafos de dependência e o desenvolvimento de um processo de comparação entre grafos mais otimizados;
2. Elaboração de um processo de extração das características mais relevantes dos grafos de dependência, o que possibilita a diminuição das estruturas armazenadas;
3. A construção de um algoritmo de comparação entre grafos de dependência, que tire proveito do modelo de classificação de vértices proposto para otimizar o

processo de comparação entre os grafos e melhorar a identificação do nível de similaridade entre eles.

1.4. Contribuições

A partir dos objetivos definidos na seção anterior, este trabalho oferece como principais contribuições:

1. **Classificação de Vértices:** Um modelo de classificação para diferenciar os tipos de vértices encontrados em um grafo de dependência. A principal vantagem desta diferenciação entre tipos de nós é eliminar, no processo de comparação, momentos onde vértices de natureza distinta seriam comparados uns com os outros, o que impede etapas de processamento ineficientes e otimiza o processo de comparação.
2. **Seleção de Elementos do Grafo**: Um método para seleção de componentes estruturais mais relevantes pertencentes a um grafo de dependência, gerado a partir de códigos executáveis. Tirando proveito do modelo de classificação de vértices, percebeu-se que um tipo particular de vértice está diretamente relacionado à maioria dos processos de tomada de decisão dos códigos modelados através dos grafos de dependência estudados nesta pesquisa. Em função disso, intuiu-se que todos os vértices associados estruturalmente a esses tipos de vértices seriam os mais relevantes para um processo de identificação. A partir disto, foi desenvolvida uma metodologia que, através do uso de estruturas virtuais, elimina os elementos estruturais de menor relevância no grafo de dependência, o que possibilita a geração de resultados mais estáveis e mais eficientes.
3. **Heurística para Comparação de Grafos**: Uma nova heurística de comparação entre grafos de dependência é proposta de forma a tirar proveito tanto do processo de classificação de vértices, como das características de ordenação topológica inerentes aos grafos de dependência, gerados a partir de programas executáveis. Isto possibilitou a geração resultados mais estáveis e precisos.
4. **Arquitetura de Identificação de *Malware***: Uma arquitetura automatizada para detectar o nível de similaridade entre códigos metamórficos executáveis. Finalmente, através da integração de todas as ferramentas e processos

desenvolvidos nesta pesquisa, é proposto um *framework* que integra os resultados individuais e culmina em uma arquitetura que contempla todo o processo de identificação de códigos maliciosos metamórficos, gerando uma nova abordagem que pode ser empregada tanto por pesquisadores como por empresas que atuam no combate à códigos maliciosos em geral.

A seguir, são apresentadas as produções científicas produzidas até o momento e originadas a partir dos resultados obtidos neste trabalho:

- Martins, G., Freitas, R. De, & Souto, E. (2014), "Estruturas Virtuais e Diferenciação de Vértices em Grafos de Dependência para Detecção de *Malware* Metamórfico.", XIV Simpósio Brasileiro em Segurança da Informação e Sistemas Computacionais (SBSeg 2014 - Artigos Completos / Full Papers), Belo Horizonte (MG), novembro.

- Martins, G., Freitas, R. De, & Souto, E. (2014). “Virtual Structures and Heterogeneous Nodes in Dependency Graphs for Detecting Metamorphic *Malware*”. In 33rd IEEE International Performance, Computing, and Communications Conference (IPCCC 2014). Austin, TX, EUA.

- Martins, G., Freitas, R. De, & Souto, E. (2015). “Processo de Detecção de Malware Metamórficos baseado em Grafos de Dependência, Diferenciação de Vértices e inclusão de Estruturas Virtuais”. Pedido de Patente. (AGUARDANDO ANÁLISE).

- Martins, G., Freitas, R. De, & Souto, E. (2016). “Diferentianting Vertex on Dependency Graphs to Identify Methamorphic Malware”. In IEEE Security & Privacy. (EM ELABORAÇÃO).

Além destas publicações, o autor desta pesquisa teve a oportunidade de contribuir em outras publicações, utilizando para isto os conhecimentos agregados durante a execução deste trabalho:

- Cozzolino, M. Martins, G. Souto, E. e Deus, F. (2012). “Detecção de Variações de *Malware* Polimórfico por Meio de Normalização de Código e Identificação de Subfluxos”, Anais do XII Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais. Curitiba. Sociedade Brasileira de Computação, p. 30-43.

- Rocha, T. ; Martins, G. B. ; Souto, E. (2013). “ETSSDetector: uma ferramenta para detecção automática de vulnerabilidades de *Cross-Site Scripting* em aplicações web”. In: Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais, 2013, Manaus. Anais do XIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais. Porto Alegre: Sociedade Brasileira de Computação, v. 1. p. 58-71.
- Barbosa, K. Souto, E. Feitosa, E., Martins, G. (2014). "Identificação e Caracterização de Comportamentos Suspeitos Através da Análise do Tráfego DNS.", XIV Simpósio Brasileiro em Segurança da Informação e Sistemas Computacionais (SBSeg 2014 - Artigos Completos / Full Papers), Belo Horizonte (MG), novembro, 2014.
- Barbosa, K. Souto, E. Feitosa, E., Martins, G. (2014). "*Botnets*: Características e Métodos de Detecção Através do Tráfego de Rede". XIV Simpósio Brasileiro em Segurança da Informação e Sistemas Computacionais (SBSeg 2014 - Minicursos), Belo Horizonte (MG), novembro.

1.5. Estrutura da Tese

Além desta introdução, compõem esta tese os seguintes capítulos:

O Capítulo 2 apresenta o problema do metamorfismo de códigos maliciosos, fornecendo detalhes a respeito do seu funcionamento e a razão pela qual as técnicas tradicionais de detecção não são efetivas. Este capítulo também apresenta uma visão geral a respeito de diversas técnicas alternativas empregadas na identificação de códigos metamórficos, além de apresentar também uma breve revisão do estado da arte.

O Capítulo 3 apresenta a arquitetura para detecção de códigos metamórficos executáveis baseada na construção e comparação de grafos de dependência proposta neste trabalho. É apresentada uma visão geral da arquitetura, com uma descrição de todo o processo de extração dos grafos de dependência, a introdução da diferenciação dos vértices e seu uso na identificação dos componentes mais relevantes do grafo, o que possibilita o aprimoramento do processo de redução e eliminação de componentes de natureza metamórfica.

O Capítulo 4 apresenta o protocolo experimental aplicado para avaliar a metodologia proposta, detalhando os experimentos e discutindo os resultados obtidos pela introdução da diferenciação de vértices do processo de redução aprimorado e na identificação do nível de similaridade entre grafos de dependência.

O Capítulo 5 apresenta as conclusões desta pesquisa, com destaque para os pontos positivos e as limitações identificadas durante a análise dos resultados, além de um breve comentário à respeito de trabalhos futuros que podem contribuir para o aprimoramento da metodologia proposta e de seus resultados.

Capítulo 2

Fundamentos Teóricos e Trabalhos Relacionados

Este capítulo introduz a fundamentação teórica necessária para o entendimento desta proposta de tese. O capítulo inicia com uma breve explanação sobre as técnicas tradicionais de detecção de códigos maliciosos, explicando a seguir como o processo de metamorfismo de código é empregado para ludibriar esta detecção. Por fim, o capítulo apresenta uma visão geral das técnicas empregadas para tratar o metamorfismo de código e o estado da arte neste campo de pesquisa.

2.1. Técnicas Tradicionais de Detecção

Como salientado nos trabalhos de Yin [3], Wurzinger et al.[4], Griffin et al.[5], Jeong[19] e Colajanni et al.[6] [7], para ser efetivo como ferramenta de identificação, os programas antivírus se baseiam normalmente em duas técnicas distintas: 1) na consulta de assinaturas, que consistem de uma sequência de caracteres que caracterizam o programa como um *malware* e que são procuradas no corpo do programa em análise, durante o processo de varredura; e 2) na utilização de heurísticas que se baseiam no monitoramento de modificações no registro do sistema, em certas bibliotecas ou interfaces do sistema.

Assinaturas são definidas por Karin [30] e Newsome et al.[31] como sendo "*uma sequência de bytes extraída de do corpo de um malware específico*". Uma assinatura é gerada tomando-se uma parte do código que seja única para aquele *malware* em específico e que tenha pouca probabilidade de ocorrer em outros programas. Normalmente, as assinaturas são armazenadas em tabelas *hash*, de forma a garantir que o acesso seja o mais ágil possível. Para um melhor entendimento de como assinaturas são criadas, um exemplo de uma assinatura de código é apresentado na Tabela 2.1.

Tabela 2.1 - Exemplo de assinatura extraída de um código.

Opcode	**Código *Assembly***
C7060F000055	mov [esi], 0x5500000F
C746048BEC5151	mov[esi+0004],0x5151EC8B
Assinatura: C7060F000055C746048BEC5151	

Fonte: Notoatmodjo[32].

Para Griffin et al.[5], o emprego de técnicas baseadas em assinaturas são mais populares, devido à baixa geração de falsos positivos. Contudo, os autores destacam a necessidade contínua da distribuição de novas atualizações, visto que para cada nova versão de um mesmo *malware* deve ser gerada uma nova entrada no banco de dados de assinaturas, o que leva também a bases de assinatura cada vez maiores. Segundo Hsiao et al.[33], apesar de populares, assinaturas tradicionais não são capazes de identificar ataques oriundos de novos *malware* ou mesmo de novas versões de *malware* já mapeados.

Wurzinger et al. [4] destacam que, em função da grande velocidade com a qual novos *malware* ou suas variações são desenvolvidos, a segurança de computadores baseada na instalação de programas tradicionais de antivírus, não é suficiente. A velocidade com a qual estes programas são atualizados não tem sido compatível com a demanda atual. Em função do montante de informação que deve ser continuamente analisado, é impraticável depender do modelo tradicional de identificação e rastreio, baseado na coleta ou submissão de programas suspeitos, por parte da comunidade de usuários, e na análise manual dos trechos de código por engenheiros de segurança. Após considerações semelhantes, Newsome e Song [34] afirmam que um processo automatizado de geração de assinaturas é fundamental para garantir que as bases de assinaturas se mantenham atualizadas, permitindo aos antivírus obter sucesso no seu processo de detecção.

Mesmo com um esforço contínuo de atualização das bases de assinaturas, Yagi et al. [18] mostram que as técnicas tradicionais utilizadas para a criação dos padrões de assinaturas em ferramentas comerciais falham na detecção de *malware*. Uma análise realizada com um conjunto de ferramentas comerciais mostraram que as taxas de detecção variam de 35% a 74% de sucesso, conforme apresentado na Tabela 2.2. Em uma avaliação similar, Kim e Moon [15] relatam um experimento onde 41 ferramentas

de detecção de *malware* foram testadas contra cinco *malware* metamórficos, obtendo taxas de sucesso que variaram de 0% (zero) até 85%.

Desta forma, fica claro que a busca de novas soluções capazes de lidar com estas ameaças é necessária para garantir um nível de segurança aceitável para a maioria dos sistemas computacionais.

Tabela 2.2 - Taxas de detecção de vírus avaliadas.

Tipos de programas antivírus	*Taxas de Detecção (%)*
Software_A	41
Software_B	57
Software_C	34
Software_D	74
Software_E	38
Software_F	35

Fonte: Yagi et al. [18]

2.2. Escondendo a Identidade de um *Malware*

Com a mudança gradual para um modelo com motivações financeiras, os desenvolvedores de códigos maliciosos passaram a dar uma maior importância à maximização do tempo em que suas criações passam incólumes pelas ferramentas de detecção, buscando com isto aumentar os seus lucros com a exploração dos recursos disponíveis nos sistemas invadidos.

Tendo por meta a contaminação de outros sistemas com versões modificadas de si mesmo [32], diversas técnicas foram desenvolvidas para que o sistema tradicional de identificação por assinaturas pudesse ser ludibriado, dando origem a códigos maliciosos furtivos que procuram ocultar sua presença dentro dos sistemas atacados. Podemos destacar cinco fases distintas na evolução dos códigos maliciosos furtivos [35]:

1. **Sem comportamento furtivo** - onde os códigos maliciosos, criados principalmente para demonstrar as capacidades técnicas de seus desenvolvedores, não possuíam qualquer característica destinada a evadir os procedimentos de identificação, sendo facilmente identificados pela técnica de

assinaturas.

2. **Criptografia -** onde o corpo principal do *malware* é criptografado por uma chave aleatória e um componente estático é empregado para restaurar o código original, no momento em que o *malware* é ativado.
3. **Oligomorfismo -** quando o componente criptográfico é escolhido aleatoriamente dentro de um conjunto pré-definido, o que possibilita aumentar a variabilidade do corpo de código gerado por este método.
4. **Polimorfismo** - quando técnicas de ofuscação de código passaram a ser aplicadas ao componente criptográfico, deixando disponível ao *malware* uma fonte virtualmente inesgotável de componentes criptográficos.
5. **Metamorfismo** - onde o processo de criptografia foi totalmente abandonado em detrimento à aplicação de técnicas de ofuscação de código em todo o corpo do *malware*, o que possibilitou a geração de versões de código distintas, mas que continuam plenamente funcionais, eliminando qualquer necessidade de reconstrução do código original.

Outra forma de olhar para este processo evolutivo nos permite dividir códigos maliciosos furtivos em apenas dois grupos: a) aqueles que dependem de um componente criptográfico, responsável tanto pela ocultação como pela reconstrução do código malicioso; e b) aqueles que empregam técnicas de ofuscação de código para alterar sua apresentação, sem com isso perder suas funcionalidades originais. Estes dois grupos e suas características mais relevantes, são apresentados com mais detalhes nas seções seguintes.

2.3. Criptografia, Oligomorfismo e Polimorfismo

Desde os primeiros *malware* que empregaram criptografia, ou suas variações posteriores de oligomorfismo e polimorfismo, a estrutura básica deste tipo de código furtivo envolve dois componentes distintos, o motor criptográfico e o corpo criptografado do *malware* propriamente dito [35] [36] [37].

O motor criptográfico tem como principal responsabilidade a restauração do corpo criptografado ao seu estado original, para que o *malware* possa agir de acordo com os objetivos para o quais foi construído. Outra responsabilidade do motor criptográfico é a geração aleatória de uma nova chave criptográfica e a criação de um novo corpo criptografado, no momento em que ocorre o processo de propagação.

Desta forma, são geradas diferentes versões do *malware*, o que torna incompatível com outras instâncias qualquer assinatura gerada a partir do corpo criptografado de uma instância em particular.

Entretanto, como os primeiros *malware* a empregar esta técnica faziam uso de um único motor criptográfico, assinaturas geradas a partir desde componente eram suficientes para identificar estes códigos maliciosos.

Para fugir desta limitação, a próxima geração de *malware* passou a fazer uso de um conjunto de motores criptográficos distintos, que eram escolhidos aleatoriamente no momento que o processo de propagação ocorria, dando origem à técnica de furtividade conhecida como Oligomorfismo. Isto aumentou o potencial de códigos distintos que podiam ser gerados a partir de um mesmo código malicioso. Entretanto, a técnica perdia sua efetividade a partir do momento que todos os componentes de criptografia eram mapeados.

A solução para esta limitação foi o emprego de técnicas de ofuscação de código que eram responsáveis por alterar a codificação interna do motor criptográfico a cada nova instância de *malware* criada, deixando à disposição deste tipo de *malware* uma quantidade infinita de motores criptográficos que já não compartilhariam uma assinatura característica entre si. Como estes componentes criptográficos gerados mantinham as mesmas características e funcionalidades originais, com alterações apenas na forma como estavam codificados, passou-se a identificar esta técnica de furtividade como Polimorfismo.

Apesar da técnica de polimorfismo ter o potencial para criar uma quantidade infinita de instâncias de *malware* sem quaisquer trechos de código significativamente similares, esta técnica ainda compartilha o mesmo problema de todas as suas predecessoras. A partir do momento que o corpo principal do *malware* é reconstruído na memória, este pode novamente ser facilmente identificado por uma assinatura. Assim, para explorar esta característica fundamental, as ferramentas de identificação de códigos maliciosos passaram a carregar os programas suspeitos em um ambiente emulado, também conhecido como *sandbox*, onde o código reconstruído era carregado e analisado, sem que qualquer tipo de contaminação ocorresse [37].

Percebendo que este conjunto de técnicas de furtividade tinha chegado ao seu limite e reconhecendo o potencial das técnicas de ofuscação de código que, até então,

tinham sido aplicadas apenas no motor criptográfico do código malicioso, os desenvolvedores de códigos maliciosos passaram a trabalhar com na próxima geração de *malware* furtivo: os códigos metamórficos.

Versões metamórficas de um mesmo *malware* possuem um corpo de código construído de forma que as instruções se apresentem de forma distinta do código original no qual se baseiam. Entretanto, diferente do corpo criptografado das gerações anteriores, este novo código é plenamente operacional, mantendo todas as funcionalidades originais intactas e sem a necessidade de reconstrução do código original. Como cada uma destas versões metamórficas também pode gerar novas versões, qualquer assinatura gerada de acordo com a metodologia tradicional de detecção não será capaz de identificar as novas versões metamórficas geradas.

Para um melhor entendimento de como o processo de metamorfismo atua, a seção 2.4 apresenta uma descrição das técnicas mais comuns e dos resultados que são capazes de produzir.

2.4. Metamorfismo e Ofuscação de Código

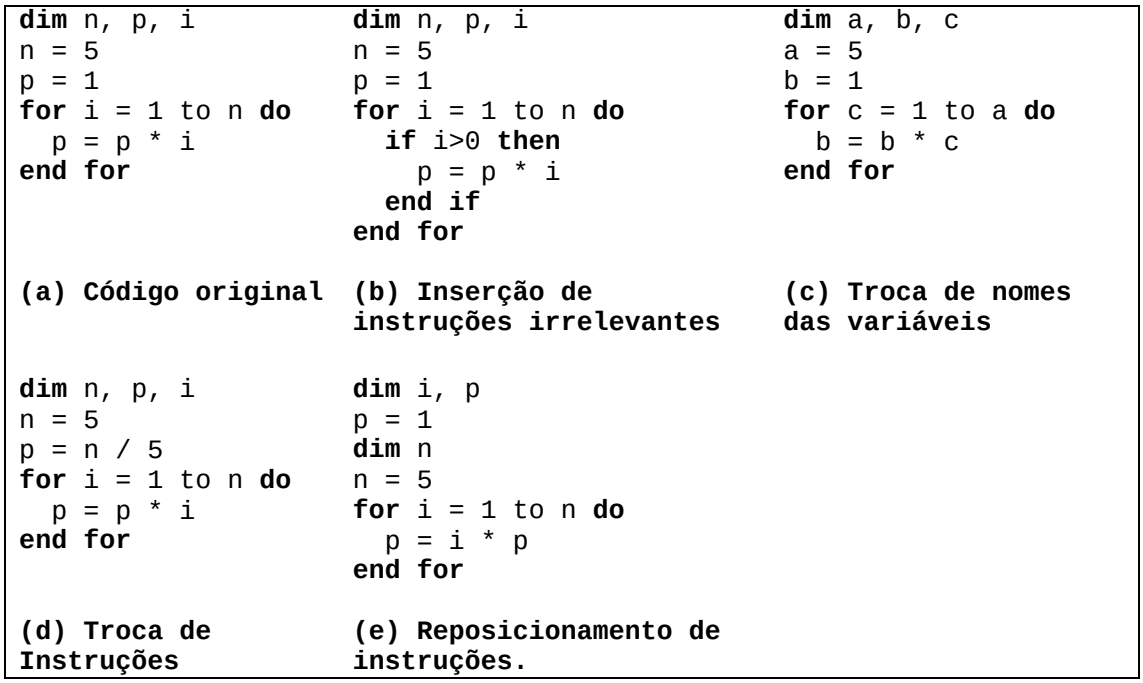
Na técnica furtiva conhecida como Metamorfismo, o motor criptográfico presente nas gerações anteriores, é substituído por um motor metamórfico, responsável por alterar a codificação do corpo do *malware*, sem que este perca qualquer uma de suas funcionalidades originais.

A efetividade desta técnica de furtividade se deve ao fato de que a maioria das assinaturas é criada com base na varredura do código binário do vírus, o que gera uma dependência das estruturas sintáticas originais e que não se mantêm nas novas versões geradas pelo motor de metamorfismo [38]. Assim, novas cópias daquele mesmo código mantêm a estrutura semântica original, mas incluem alterações sintáticas que os tornam incompatíveis com assinaturas de códigos de versões mais antigas.

Diversos trabalhos [9][23][35][37][39], apresentam algumas das técnicas metamórficas mais comuns. Alguns exemplos destas técnicas incluem: *i)* a inserção de instruções e variáveis irrelevantes (código “lixo”) que não alteram a lógica original do programa; *ii)* a alteração no nome de variáveis ou troca mútua de variáveis entre instruções diferentes; *iii)* a substituição de sequências de instruções por outras que produzam o mesmo resultado; e *iv)* a alteração na ordem de execução das instruções,

seja pelo reposicionamento de blocos de código independentes ou pelo uso de instruções de desvio de fluxo. A Fonte: Kim e Moon[15].

Figura 2.1 apresenta exemplos destas operações, em termos de instruções de alto nível [15].



```
dim n, p, i
n = 5
p = 1
for i = 1 to n do
  p = p * i
end for
```

(a) Código original

```
dim n, p, i
n = 5
p = 1
for i = 1 to n do
  if i>0 then
    p = p * i
  end if
end for
```

(b) Inserção de instruções irrelevantes

```
dim a, b, c
a = 5
b = 1
for c = 1 to a do
  b = b * c
end for
```

(c) Troca de nomes das variáveis

```
dim n, p, i
n = 5
p = n / 5
for i = 1 to n do
  p = p * i
end for
```

(d) Troca de Instruções

```
dim i, p
p = 1
dim n
n = 5
for i = 1 to n do
  p = i * p
end for
```

(e) Reposicionamento de instruções.

Fonte: Kim e Moon[15].

Figura 2.1 - Exemplos de técnicas de ofuscação de código, onde o código original (a) é alterado pela inserção de instruções irrelevantes (b), pela troca de nomes das variáveis (c), troca de instruções (d) e pelo reposicionamento de instruções (e).

2.4.1. Inserção de Códigos Irrelevantes

Uma das técnicas mais simples para modificar a apresentação original do código trata-se da inserção de instruções irrelevantes ao processamento dos trechos de código onde são inseridas, mudando assim a sua aparência original, sem com isso inserir qualquer modificação relevante no estado do programa.

Um exemplo de ação de destas técnicas é apresentado por Karnik et al.[39] na Tabela 2.3. Através da inserção de alguns NOPS, a assinatura relacionada àquele trecho de código é alterada de 5150 5B8D 4B38 50E8 0000 0000 5B83 C31C FA8B 2B5B, para 5190 505B 8D4B 3850 90E8 0000 0000 5B83 C31C FA90 8B2B 5B, frustrando um processo de identificação que executasse apenas uma comparação direta. Outros exemplos de instruções irrelevantes são apresentados na Tabela 2.4.

Entretanto, teste tipo de técnica pode ter seus efeitos anulados, caso a ferramenta de identificação ignore totalmente todas as instruções que forem identificadas como inócuas [37]. Assim, uma evolução desta técnica passou a incluir conjuntos de instruções que produzem alterações no estado original do programa modificado, mas que tem essas alterações anuladas por outro conjunto de instruções inseridas a seguir, restaurando o estado do programa à sua normalidade antes que o resultado do processamento possa ser impactado[35]. Exemplos destes grupos de instruções são apresentados na Tabela 2.5.

Tabela 2.3 – Exemplo de metamorfismo empregando inserção de código lixo.

Código original		***Código modificado***	
Opcodes	***Assembly***	***Opcodes***	***Assembly***

51
50
5B
8D4B38
50
E800000000
5B
83C31C
FA
8B2B
5B
	push ecx
	push eax
	pop ebx
	lea ecx, [ebx + 38h]
	push eax
	call 0h
	pop ebx
	add ebx, 1Ch
	Cli
	mov ebp, [ebx]
	pop ebx
		51
		90
		50
		5B
		8D4B38
		50
		90
		E800000000
		5B
		83C31C
		FA
		90
		8B2B
		5B
			push ecx
			Nop
			push eax
			pop ebx
			lea ecx, [ebx + 38h]
			push eax
			Nop
			call 0h
			pop ebx
			add ebx, 1Ch
			Cli
			Nop
			mov ebp, [ebx]
			pop ebx

Assinatura: 51505B8D4B3850E8000000005B83C31CFA8B2B5B		***Assinatura***: 5190505B8D4B385090E8000000005B83C31CFA908B2B5B	

Fonte: Karnik et al.[39].

Tabela 2.4 – Exemplos de instruções inócuas que poder alterar a assinatura de um trecho de código, sem interferir com seu o comportamento original.

Instruções
ADD *Registrador*, **0**
MOV Registrador, *Registrador*
OR *Registrador*, **0**
AND *Registrador*, **-1**

Fonte: Rad et al. [35].

Tabela 2.5 – Exemplos de instruções que modificam o estado do programa, mas que são anuladas a seguir, tornando o conjunto irrelevante.

Instruções modificadoras	***Anulada por***
PUSH *Registrador*	**POP** *Registrador*
ADD *Registrador*, **3**	**SUB** *Registrador*, **1** **SUB** *Registrador*, **2**

É interessante destacar que as instruções que se anulam não precisam necessariamente vir aos pares ou mesmo uma logo após a outra, o que dificulta ainda mais a identificação desta técnica.

2.4.2. Troca de Variáveis entre Instruções

Outra técnica simples utilizada pelos atacantes para mudar a aparência do código corresponde a troca mútua variáveis de memória entre instruções [37] [39] [35], sem que qualquer outro tipo de modificação seja aplicada.

Para programas executáveis, este processo é adaptado para incluir também a troca de registradores entre as instruções. A Tabela 2.6 apresenta um exemplo deste tipo de alteração. Neste exemplo é possível notar que uma comparação direta entre os trechos alterados não produziria uma identificação positiva.

Uma forma comum de lidar com esta técnica é a introdução de assinaturas que ignoram as porções do código associadas aos registradores e que usam apenas os códigos de instruções como base de identificação. Entretanto, quando a troca de variáveis e registradores é combinada com outras técnicas de ofuscação de código, a quantidade de modificações introduzidas em cada versão inviabiliza completamente o uso de técnicas de identificação baseadas em assinaturas.

Tabela 2.6 – Exemplo de metamorfismo empregando troca mútua de registradores (variáveis) entre operações.

Código original		***Código modificado***	
Opcode	***Assembly***	***Opcode***	***Assembly***

5ª
BF04000000
8BF5
B80C000000
81C288000000
8B1A
899C8618110000
	pop edx
	mov edi,0004h
	mov esi,ebp
	mov eax,000Ch
	add edx,0088h
	mov ebx,[edx]
	mov [esi+eax*4+00001118],ebx
		58
		BB04000000
		8BD5
		BF0C000000
		81C088000000
		8B30
		89B4BA1811000
			pop eax
			mov ebx,0004h
			mov edx,ebp
			mov edi,000Ch
			add eax,0088h
			mov esi,[eax]
			mov [edx+edi*4+00001118],esi

Assinatura: 5ABF040000008BF5B80C00000081C2880000008B1A899C8618110000		***Assinatura*:** 58BB040000008BD5BF0C00000081C0880000008B3089B4BA18110000	

Fonte: Karnik et al.[39].

2.4.3. Troca de Instruções por outras de Resultado Equivalente

Outra forma muito comum de ocultar a identidade de um programa é a substituição de uma instrução ou um conjunto de instruções por outras que produzam o mesmo

resultado das originais [35] [37] [39].

A Tabela 2.7 mostra um exemplo de uma lista de instruções diferentes que são capazes de produzir o mesmo resultado. Neste exemplo, todas as instruções podem ser usadas para zerar o conteúdo de um registrador específico. Um exemplo da aplicação desta técnica em um trecho de código específico é mostrado da Tabela 2.8.

Tabela 2.7 – Exemplos de instruções que produzem o mesmo resultada da original.

Instrução original	*Pode ser trocada por*
MOVE *Registrador*, 0	**XOR** *Registrador*, *Registrador* **SUB** *Registrador*, *Registrador* **AND** *Registrador*, **0**

Tabela 2.8 - Exemplo de metamorfismo empregando trocas de trechos de instruções por outros de mesmo efeito [39]**.**

Código original:		***Código modificado:***	
Opcode	***Assembly***	***Opcode***	***Assembly***
55	push ebp	55	push ebp
8BEC	mov ebp, esp	54	push esp
8B7608	mov esi, dword ptr [ebp + 08]	5D	pop ebp
85F6	test esi, esi	8B7608	mov esi, dword ptr [ebp + 08]
743B	je 401045	09F6	or esi, esi
8B7E0C	mov edi, dword ptr [ebp + 0c]	743B	je 401045
09FF	or edi, edi	8B7E0C	mov edi, dword ptr [ebp + 0c]
7434	je 401045	85FF	test edi, edi
31D2	xor edx, edx	7434	je 401045
		28D2	sub edx, edx
Assinatura: 558BEC8B760885F6743B8B7E0C09FF743431D2		***Assinatura*:** 55545D8B760809F6743B8B7E0C85FF743428D2	

Fonte: Karnik et al.[39].

2.4.4. Mudança no Posicionamento das Instruções

Esta técnica não altera as instruções originais, mas sim a posição onde estas instruções se apresentam no código [35] [37] [39]. Quaisquer instruções que sejam mutuamente independentes, não importando a ordem em que são executadas, que não interfiram umas com as outras, podem ser trocadas de posição, sem que o resultado final do código seja alterado.

Um exemplo de um trecho de código constituído deste tipo de instrução é apresentado na Tabela 2.9.

Tabela 2.9 – Exemplos de instruções que podem ter suas posições trocadas.

Ordem original	*Pode ser trocada por*
MOVE *EAX*, 0 **PUSH** *ECX* **ADD** ESI, EBX	**ADD** ESI, EBX **PUSH** *ECX* **MOVE** *EAX*, 0

Apesar de não ser considerada uma atividade trivial, o desenvolvedor do código malicioso pode fazer um mapeamento das relações de dependência que existem entre as instruções do código. Assim, qualquer cadeia de instruções que não sejam dependentes entre si, podem ter suas posições trocadas, alterando significativamente a codificação original.

Outra forma de produzir este mesmo efeito é através da utilização de várias instruções de desvio incondicional. Neste caso, as instruções originais podem ser colocadas virtualmente em qualquer posição do código, mesmo que exista uma relação de dependência entre elas. Isto é possível, pois as instruções de desvio incondicional irão garantir a execução das instruções na ordem correta.

Esta segunda forma de rearranjo das instruções tem o potencial de gerar uma quantidade muito grande de variações de um mesmo código (***n!*** onde ***n*** representa a quantidade de instruções de desvio incondicional presentes no código). Entretanto, o desenvolvedor de códigos maliciosos deve tomar cuidado com a quantidade de instruções de desvio incondicionais presentes, já que um código que possua uma quantidade muito significativa de instruções desta natureza seria considerado automaticamente como suspeito.

2.5. Abordagens para Detecção de *Malware* Metamórficos

Para lidar com o problema do metamorfismo em *malwares*, um conjunto de diferentes abordagens foram propostas. Neste trabalho, classificamos estas abordagens em quatro grupos: 1) técnicas baseadas em análise estática do código; 2) técnicas de mineração de dados, 3) técnicas baseadas na análise comportamental, e 4) técnicas de normalização de código. Embora existam metodologias que utilizam apenas uma destas técnicas, é habitual combinar mais do que uma para delas para que os resultados desejados sejam

obtidos.

2.5.1. Análise Estática do Código

Neste primeiro grupo, as abordagens propostas pretendem identificar o *malware*, analisando suas características estruturais, mas com foco na semântica ou outras características peculiares, tais como fluxo de dados, chamadas de sistema, dependência de instruções de código, uso de motores (*engines*) criptográficos, ou outra característica particular que seja necessária para uso da metodologia empregada. No entanto, a escolha de qual característica em particular será investigada, dependerá das etapas restantes de cada metodologia.

Outra forma comum de análise estática é o levantamento dos fluxos de controle ou de dados, para a coleta de informações sobre a organização estrutural do código. A principal vantagem desta abordagem está no fato de pode ser usada sem a necessidade de carregar o código malicioso para a memória, diminuindo os riscos de contaminação do ambiente de pesquisa. Entretanto, esta técnica possui o inconveniente de depender da extração eficaz das características selecionadas.

2.5.2. Mineração de Dados

Este grupo de abordagens se concentra no uso de ferramentas estatísticas para modelar características comuns compartilhadas por códigos maliciosos, possibilitando assim a identificação de grupos ou famílias de *malware*. Para que as informações relevantes sejam identificadas, as técnicas de mineração de dados necessitam ter acesso a uma grande coleção de códigos distintos, para que o modelo de detecção construído seja capaz de modelar tendências e características presentes nos códigos maliciosos.

Finalmente, por se tratar de um processo automatizado, a mineração de dados é capaz de identificar tendências e características que poderiam passar despercebidas, se o trabalho de coleta fosse executado por um analista humano. Entretanto, a qualidade dos modelos construídos dependerá muito da representatividade os conjuntos utilizados para treinar o classificador.

2.5.3. Análise Comportamental

Esta abordagem se concentra em identificar os resultados ou as consequências que podem ser encontradas em um sistema contaminado, derivadas das ações executadas

pelos códigos maliciosos. Isto se justifica pela suposição de que, mesmo escritos de forma diferente, programas com o mesmo objetivo final podem apresentar comportamentos similares. Desta forma, os modelos construídos a partir desta técnica não seriam utilizados para identificar um único *malware*, mas toda uma família de códigos maliciosos.

A extração de características comportamentais do programa em análise pode ser feita tanto através do monitoramento de sua execução, como através de uma análise estrutural do código que o constitui. A maior vantagem desta abordagem é a falta de necessidade de se preocupar com as técnicas de ofuscação que porventura tenham sido utilizadas no código do *malware*, visto que o que está sendo monitorado são as ações e os resultados gerados, e não a forma como os mesmos foram codificados.

Entretanto, como o *malware* precisa estar agindo ativamente para que os resultados de suas ações possam aparecer, este procedimento não é efetivo para prevenir a contaminação de um sistema, sendo utilizado para identificar desvios no comportamento normalmente apresentado pelo mesmo, e que caracterizariam que este sistema se tornou contaminado.

Outra limitação identificada é a falta de efetividade para lidar com um sistema que se encontram contaminado antes do início da construção do modelo comportamental, pois o modelo construído consideraria a presença do *malware* algo normal. Modelos comportamentais baseados na análise estrutural não compartilham desta limitação, mas terão as mesmas características positivas e negativas das técnicas de análise estática de código.

2.5.4. Normalização de Código

Esta última categoria corresponde às abordagens que concentram seus esforços em reverter os efeitos de técnicas de ofuscação, permitindo a correta identificação do *malware* original. Isto leva a uma abordagem capaz de identificar qualquer versão de um mesmo *malware* metamórfico através de um único padrão de identificação. Outra situação comum é a utilização de dessas abordagens para simplificar o código analisado e facilitar o trabalho de uma etapa seguinte na metodologia de identificação do *malware*.

A efetividade desta abordagem depende da sua capacidade de desfazer o metamorfismo de código e de quão similar o código reconstruído será ao modelo de

referência adotado. Infelizmente, para cada nova geração metamórfica, o processo de reversão é cada vez mais difícil de ser obtido.

2.6. Trabalhos Relacionados

As abordagens descritas na seção 2.5 são empregadas em diversos trabalhos que tratam do problema de identificação de *malware*. Além de apresentar exemplos distintos de implementação, é comum que estes trabalhos empreguem mais de uma dessas técnicas nas metodologias propostas, encadeando os resultados individuais de forma a atingir a meta principal de identificação de códigos maliciosos.

Uma primeiro exemplo apresentado por Griffin et al. [5], que propõem uma alternativa ao modelo tradicional de assinaturas, baseado em um procedimento automatizado para a análise de grupos de *malware* previamente identificados, criando um conjunto mínimo não linear de sequências de *bytes* de tamanho ***n***, baseado na probabilidade de um símbolo vir após uma sequência qualquer de símbolos. A Fonte: Griffin et al. [5].

Figura 2.2 mostra a avaliação de probabilidade de caráter "*e*" segue a *string* "*abcd*", onde *count(s)* é o número de ocorrências das sequências de *bytes* no conjunto de treinamento.

$$p(\mathrm{e}|\mathrm{abcd}) = \frac{count(\mathrm{abcde})}{count(\mathrm{abcd})} * (1 - \epsilon(count(\mathrm{abcd}))) + p(\mathrm{e}|\mathrm{bcd}) * \epsilon(count(\mathrm{abcd}))$$

Fonte: Griffin et al. [5].

Figura 2.2 - Exemplo de avaliação de probabilidade de um conjunto de caracteres.

Esta técnica faz uso de três heurísticas distintas para seleção das melhores sequências de identificação, que serão então conhecidas como *assinaturas string*. Sequências candidatas são descartadas pelas duas primeiras heurísticas quando, de acordo com um modelo pré-definido, tiverem alta probabilidade de serem encontradas em programas comuns, uma vez que se originam do uso de bibliotecas de funções. Finalmente, a última seleção é feita baseada no fato de que uma assinatura candidata associada com um conjunto de *malware* similares tem uma chance menor de gerar falsos positivos, desde que existe uma boa probabilidade desta assinatura ser derivada de uma sequência de código única dentro de uma família de *malware*. Segundo os autores, seria possível melhorar o procedimento pela geração de assinaturas candidatas baseada em múltiplos trechos não consecutivos de código, mas a sobrecarga

computacional derivada disto não seria irrelevante.

A utilização de autômatos finitos, apresentada em Jacob et al. [13], é outra técnica utilizada para identificar o comportamento apresentado por códigos maliciosos através de um modelo construído com base no fluxo de dados associado ao seu comportamento. Um autômato finito é usado para modelar uma sequência de chamadas a funções do sistema operacional, caracterizando o comportamento de uma ação maliciosa. O procedimento proposto tenta identificar os programas suspeitos através do monitoramento do fluxo de dados e da comparação com autômatos previamente construídos para modelar o comportamento apresentado por um *malware*. Exemplos comuns deste tipo de comportamento suspeito são os processos de *duplicação*, onde um programa cria novas cópias de si mesmo dentro de uma mesma máquina, e de *propagação*, onde um programa envia cópias de si mesmo para outros computadores em um ambiente de rede. A Fonte: Jacob et al. [13].

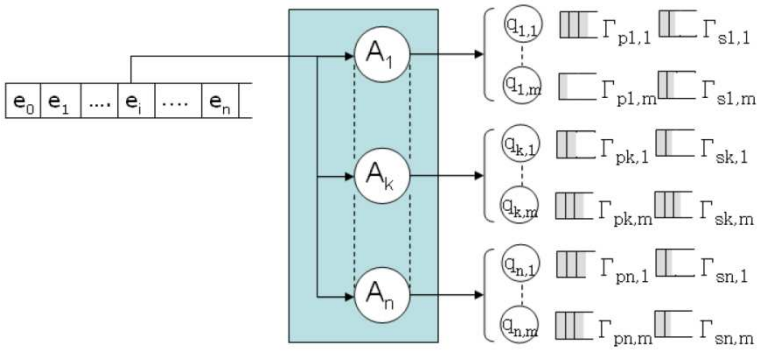
Figura 2.3 apresenta a descrição comportamental da ação *Propagate* (propagação) de um *malware*. Neste exemplo, a ação de propagação é modelada pelas primitivas de *Open* (abrir um arquivo para leitura ou escrita), *Read* (ler o conteúdo do arquivo que será infectado) e *Transmit* (transferir o corpo do *malware* para o arquivo que será infectado). A primitiva *Transmit*, é ainda detalhada como formada principalmente por uma primitiva *Write* (a escrita propriamente dita do conteúdo do *malware*) que pode ou não ser precedida de uma primitiva *Format* (quando o código malicioso precisa passar por um processo de transformação ou mutação, antes de ser efetivamente copiado para o seu destino). Fonte: Jacob et al. [13].

Figura 2.4 mostra um autômato paralelo para detecção comportamental. A maior limitação deste método é a necessidade da criação manual destes autômatos, o que requer um grande conhecimento a respeito do fluxo de dados e do comportamento apresentado tanto por códigos maliciosos como por programas benignos, o que não é algo tão simples de se obter.

$$
\begin{array}{lll}
(i) \quad <Propagate> & ::= & <Open><Read><Transmit> \\
 & | & <Read><Open><Transmit> \\
\{ \quad \ldots & & \\
\quad (<Propagate>.\text{srcTp} & = & this \\
\quad \vee <Propagate>.\text{srcId} & = & <Duplication>.\text{targId}) \\
\quad \ldots & & \\
\quad <Propagate>.\text{targTp} & = & obj\_com \\
\quad \ldots & & \} \\
(ii) \quad <Transmit> & ::= & <Format><Write> \\
 & | & <Write>
\end{array}
$$

Fonte: Jacob et al. [13].

Figura 2.3 - Exemplo de descrição do um comportamento malicioso de propagação.



Fonte: Jacob et al. [13].

Figura 2.4 - Autômato finito paralelo para detecção comportamental.

Rieck et al. [8] propõem uma metodologia para identificar *malware* baseados em amostras de códigos maliciosos previamente identificados e na construção de modelos capazes de classificar novas versões de *malware* pertencentes a uma mesma família. De modo geral, a metodologia tenta responder dois questionamentos: i) o *malware* analisado pertence a uma família conhecida ou é o primeiro representante de uma família completamente nova? e, ii) quais características comportamentais são realmente importantes para ser fazer a distinção entre as várias famílias de *malware*?

Para responder estas questões, a metodologia proposta busca mapear estes comportamentos baseados em amostras de códigos maliciosos previamente identificados e na construção de modelos capazes de classificar novas versões de *malware* pertencentes a uma mesma família. As principais etapas desta metodologia são:

a) Coleta de programas suspeitos através do uso de *honeypots* [40]–[43] e *spam-traps*, para uma tentativa inicial de identificação com o uso de ferramentas de antivírus e para o levantamento dos padrões comportamentais descobertos

durante a execução deste programa em um ambiente *sandbox*;

b) Construção de um modelo distinto para cada família de *malware* previamente mapeada, proporcionando uma forma de avaliar as informações coletadas e oferecer os meios para estabelecer a qual família o código analisado pertence;

c) Aprimoramento dos modelos construídos através da utilização de pesos associados a cada característica coletada na primeira etapa, para destacar as características peculiares encontrada em cada família de malware.

A Fonte: Rieck et al. [8].

Figura 2.5 apresenta um exemplo de cinco operações usadas para caracterizar as características comportamentais para a família do *worm Worm.Sality* [8]. Contudo, os seus autores deixam claro que as principais limitações do modelo proposto estão associadas à incapacidade de simular todas as possibilidades de execução do código suspeito dentro do ambiente monitorado e da incapacidade deste modelo lidar com *malware* relacionado a famílias desconhecidas.

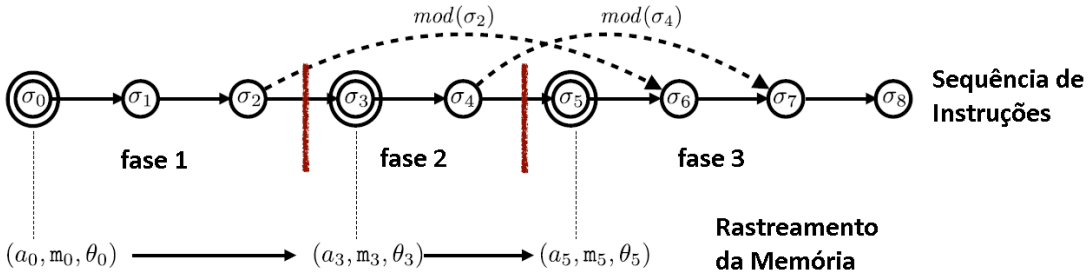
```
0.0142: create_file_2 (srcpath="C:\windows\...")
0.0073: create_file_1 (srcpath="C:\windows\...", srcfile="vcmgcd32.dl_")
0.0068: delete_file_2 (srcpath="C:\windows\...")
0.0051: create_mutex_1 (name="kuku_joker_v3.09")
0.0035: enum_processes_1 (apifunction="Process32First")
```

Fonte: Rieck et al. [8].

Figura 2.5 - Exemplo de cinco operações usadas para caracterizar comportamentos maliciosos.

Buscando construir um modelo semântico que não exija conhecimento prévio das características comportamentais de um *malware* metamórfico, Preda et al. [44] propõem uma abordagem onde, através de uma modelagem semântica, é construída uma *assinatura metamórfica*. O código metamórfico é analisado estaticamente e dividido em um conjunto de fazes distintas, correspondente a blocos de código delimitados pelo surgimento de instruções que atuam na modificação do código que será executado posteriormente. Um exemplo deste tipo de estrutura é apresentado na Fonte: Preda et al. [44].

Figura 2.6. Neste exemplo, as instruções 1 e 4 alteram as instruções 6 e 7, respectivamente.

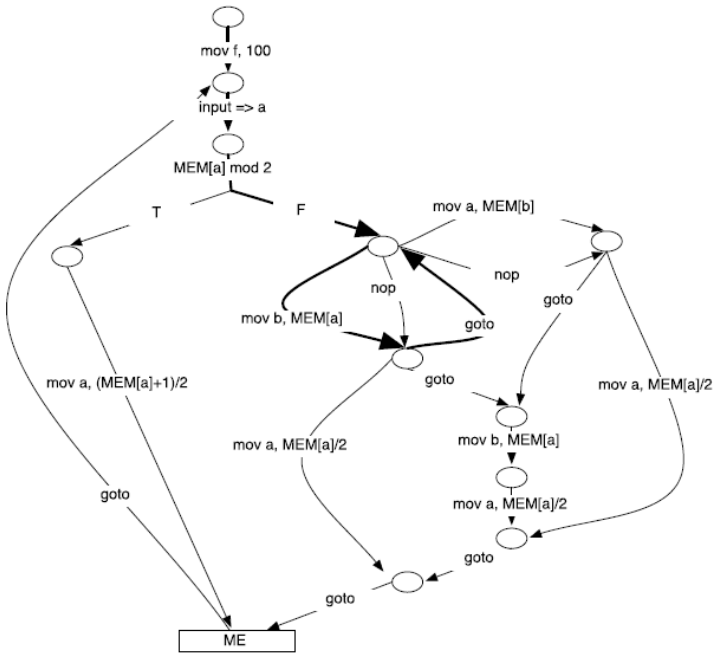


Fonte: Preda et al. [44].

Figura 2.6 – Exemplo de divisão em fases de um código metamórfico que contém instruções que alteram seu próprio código.

O modelo construído a partir desta análise cria um autômato finito destinado a detectar variações metamórficas do programa original. O processo de identificação passar a ser tratado como o reconhecimento de uma linguagem regular: caso uma instância de código seja reconhecida como um elemento válido da gramática, a identificação é positiva. Um exemplo deste tipo de autômato é apresentado na Fonte: Preda et al. [44].

Figura 2.7. Entretanto, o modelo proposto foi avaliado com base em apenas um exemplo de código malicioso. Assim, apesar dos resultados iniciais positivos, este modelo ainda necessita de maior validação.

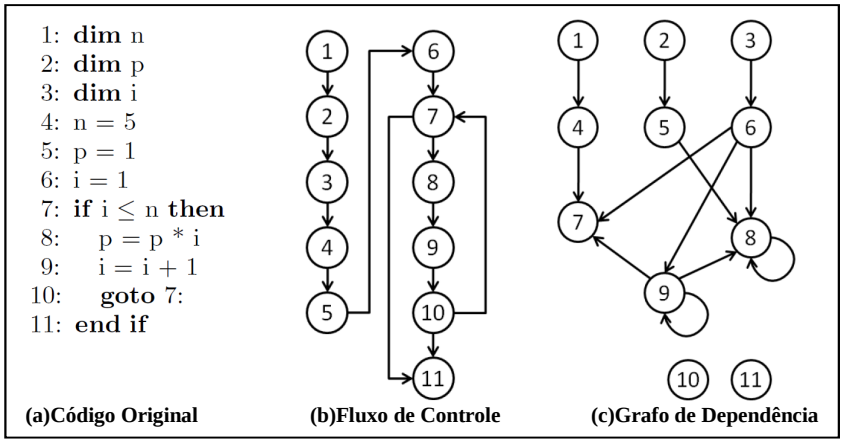


Fonte: Preda et al. [44].

Figura 2.7 – Exemplo de autômato finito para reconhecimento de código metamórfico.

Kim e Moon [15] apresentam um processo de detecção de códigos maliciosos inseridos dentro de programas script. Apesar deste tipo de programa ser executado através de um interpretador, não existe impedimento que técnicas de ofuscação de código presentes em códigos binários executáveis sejam aplicadas. Para lidar com este problema, o código suspeito é analisado para a geração de um grafo de dependência que modela as inter-relações entre cada instrução presente no código, baseadas nas variáveis que manipulam. Cada vértice é associado a uma instrução e, para cada variável manipulada, uma aresta orientada liga o vértice que representa esta instrução ao vértice correspondente à próxima instrução que estiver manipulando esta mesma variável.

A Figura 2.8 apresenta um exemplo de programa *script* e seus diagramas de controle de fluxo e grafo de dependência. Em seguida, este grafo passa por um processo de normalização que descarta vértices isolados, eliminando instruções inseridas pelas ações das técnicas de ofuscação de código, além de diminuir o tamanho do grafo original. Deste ponto em diante, o processo de detecção é baseado no problema de encontrar o máximo isomorfismo de subgrafo entre o grafo normalizado e um grafo que modela um código malicioso previamente identificado. Entretanto, como se trata de um problema *NP-Difícil*, o custo de processamento é alto, o que exige a utilização de heurísticas para a diminuição do tempo gasto neste processo.



Fonte: Kim e Moon [15].

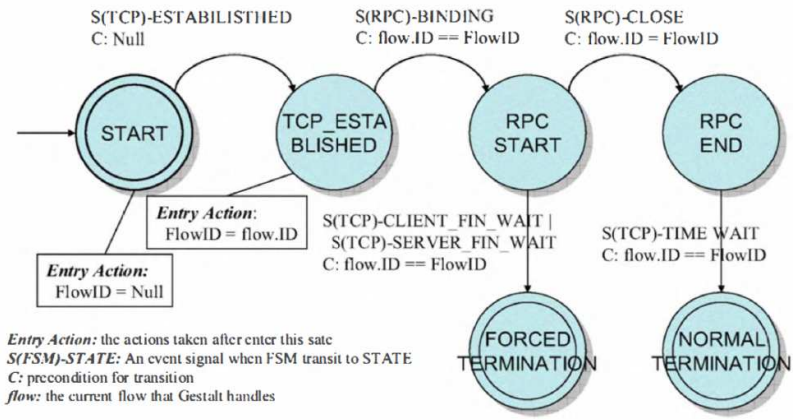
Figura 2.8 - Exemplo de código script (a) e seus respectivos fluxos de controle (b) e grafo de dependência (c).

A proposta de Hsiao et al.[33] consiste de uma técnica baseada na identificação de comportamentos anômalos dentro do tráfego da rede, através do uso de aprendizado de máquina e analisadores estatísticos. É construído, então, um modelo que representa o comportamento normal ou esperado da rede e qualquer desvio deste comportamento,

como uma alta taxa de mensagens de falhas de conexão, é encarado como uma evidência de ataque em progresso.

A Fonte: Hsiao et al.[33].

Figura 2.9 apresenta um autômato finito capaz de identificar este tipo de ataque. Assim, este método pode identificar anomalias comportamentais que são apresentadas tanto pelo atacante como pela vítima, durante um processo de invasão, quando as falhas de segurança presentes no sistema estão sendo exploradas. Segundo os autores, os modelos comportamentais constituem um conjunto de *assinaturas comportamentais* que são mais resistentes às tentativas de modificação na codificação dos códigos maliciosos, pois estão intrinsecamente associadas ao processo de exploração das falhas de segurança presentes no sistema, que independe da forma como o programa malicioso foi codificado. Entretanto, para ser efetiva, esta técnica necessita que a modelagem do comportamento normal ocorra em um ambiente livre de qualquer tipo de contaminação, pois a ação de qualquer atacante já instalado seria considerada com parte do comportamento normal, tornando-o indetectável.



Fonte: Hsiao et al.[33].

Figura 2.9 - Exemplo de um autômato finito destinado a identificar um ataque de término forçado de seção.

Um exemplo de técnica de identificação que não lida diretamente com o código, mas sim com as ações resultantes da presença de um *malware*, é apresentado por Wurzinger et al.[4]. O sistema proposto é capaz de identificar computadores previamente infectados por *botnets*, um tipo de *malware* que depende de uma comunicação com os seus proprietários, sem que para isso seja necessário de qualquer conhecimento prévio relacionado aos comandos, canais de controle ou vetor de propagação do *malware* em questão. Este sistema se baseia em um modelo de detecção que assume que todas as *botnets* recebem comandos originados de um ponto central

(conhecido como *botmaster*) e que as respostas a estes comandos têm um comportamento previsível.

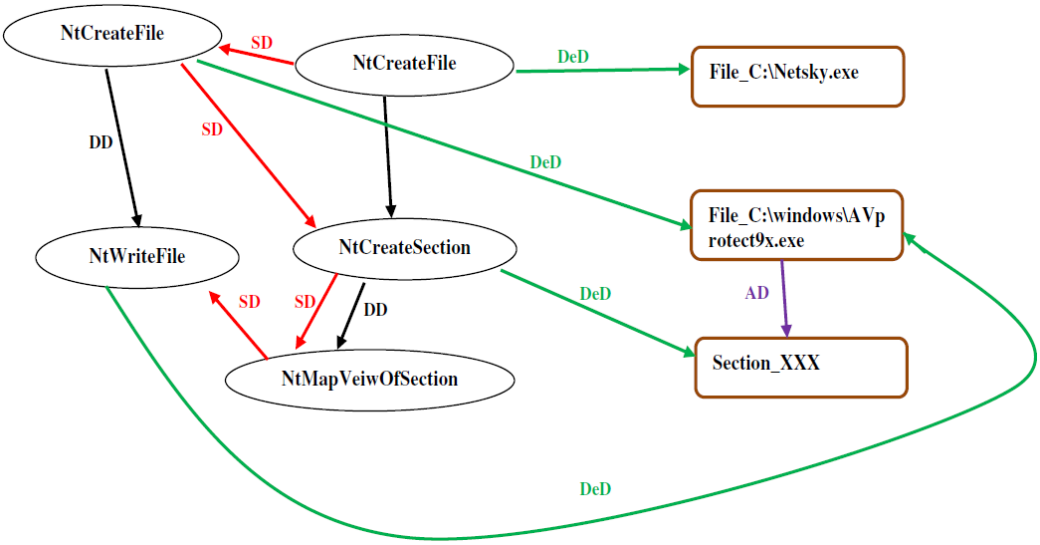
O ponto chave desta metodologia é executar o malware em um ambiente controlado e esperar que a comunicação com o *botmaster* seja iniciada. Quando isto ocorre, todas as mensagens trocadas serão coletadas e um modelo de detecção é construído, para servir de base de comparação com outras mensagens da rede, de forma a identificar novas contaminações do mesmo *malware*. A Fonte: Wurzinger et al.[4].
Figura 2.10 mostra uma lista de características de tráfego de rede usadas para identificar o comportamento de uma botnet. Entretanto, os autores alertam a respeito da possibilidade deste processo de identificação ser evitado através do atraso no envio de respostas ao *botmaster*, tornando mais difícil o estabelecimento de uma relação entre as mensagens trocadas. Outros problemas são apresentados em detalhe por Zou e Cuninghan [45].

Número de pacotes	Número de bytes no corpo que estão fora do padrão ASCII
Tamanho cumulativo dos pacotes (em bytes)	Número de pacotes UDP
Número de IPs diferentes contatados	Número de pacotes HTTP (porta de destino 80)
Número de portas diferentes contatadas	Número de pacotes SMNP (porta de destino 25)

Fonte: Wurzinger et al.[4].

Figura 2.10 - Lista de características utilizadas para identificar o comportamento de uma *botnet*.

Outra abordagem para o uso de grafos na detecção de códigos maliciosos metamórficos é proposta por Elhadi et al. [46], que trata o problema de identificação através do uso de grafos que modelam chamadas bibliotecas de APIs (Aplication Program Interfaces). Este modelo ignora completamente a forma como o programa está codificado, se concentrando em representar as relações de dependência que existem entre os diversos módulos (procedimentos e funções) que são utilizados pelo programa modelado, sejam chamadas a bibliotecas externas ou mesmo chamadas a funções do sistema operacional. Um exemplo teste tipo de grafo, também conhecido como grafo de chamada, é apresentado na Fonte: Elhadi et al. [46].
Figura 2.11.



Fonte: Elhadi et al. [46].

Figura 2.11 – Grafo de dependência de chamadas de função.

Ignorando a forma como o programa está codificado, esta proposta procura invalidar totalmente o metamorfismo de código, se concentrando em construir o seu modelo de identificação baseado na forma como os diversos módulos do programa interagem entre si e entre as funções externas utilizadas. Assim, quaisquer programas passar a ser comparados através da medição do nível de similaridade entre seus grafos de chamada. Caso o nível de similaridade obtido ultrapasse um determinado limiar, a identificação é considerada positiva. As maiores limitações deste método estão associadas ao nível de precisão do processo de extração do grafo de chamadas e ao custo computacional na execução do processo de comparação dos grafos.

2.7. Considerações Finais

Uma relação entre as metodologias apresentadas e as abordagens empregadas em cada uma delas é apresentada na Tabela 2.10. Aqui é possível perceber, de forma resumida, quais técnicas foram usadas em cada uma das metodologias destacadas nesta tabela.

A partir dos trabalhos apresentados nesta seção, é possível perceber que o problema de identificação de códigos maliciosos metamórficos ainda se encontra em aberto e que as metodologias propostas tem ainda espaço para aprimoramentos. Nos próximos capítulos serão apresentados os conceitos básicos onde esta proposta de tese se alicerça, bem como toda a metodologia de detecção proposta.

Tabela 2.10 - Resumo das metodologias e técnicas utilizadas para a identificação de códigos maliciosos metamórficos.

Metodologia	Análise de Código	Mineração de Dados	Análise Comportamental	Normalização
Griffin et al.	Sim	Sim	Não	Não
Jacob et al.	Sim	Não	Sim	Não
Rieck et al.	Não	Sim	Sim	Não
Kin e Moon	Sim	Não	Sim	Sim
Hsiao et al.	Não	Sim	Sim	Não
Wurzinger et al.	Não	Sim	Sim	Não
Preda et al.	Sim	Não	Sim	Não
Elhadi et al.	Sim	Não	Sim	Não

Capítulo 3
Estruturas Virtuais e Diferenciação de Vértices em Grafos de Dependência

Este capítulo apresenta todos os conceitos básicos que dão suporte à metodologia de identificação de *malware*, bem como apresenta uma visão geral da arquitetura proposta e uma descrição de todo o processo de extração dos grafos de dependência. A seguir são apresentados os detalhes a respeito do uso da diferenciação dos vértices no processo de identificação dos componentes mais relevantes do grafo, concluindo com a explicação de como esta diferenciação contribui com o processo de comparação entre os grafos de dependência usados para identificar um *malware* metamórfico.

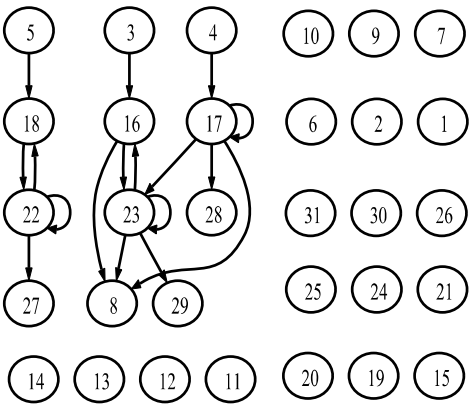
3.1. Grafos de Dependência na Identificação de Códigos Metamórficos

Grafos de dependência são grafos direcionados que representam relações de dependência de entre elementos pertencentes a uma mesma estrutura [47]. Originalmente empregados na identificação de plágios [17], os grafos de dependência também tem sido usado como estrutura base para eliminar as ações das técnicas de ofuscação em códigos maliciosos [23][43]. Nessa abordagem, o problema de se identificar o nível de similaridade entre dois programas é modelado através de um grafo direcionado, onde cada instrução do código corresponde a um vértice e, para cada variável que esta instrução manipular, uma aresta direcionada será inserida, ligando esta instrução à próxima linha de código que manipular esta mesma variável. Este procedimento é aplicado tanto a um programa suspeito de contaminação como no malware que será investigado, gerando dois grafos que serão comparados. Encontrar correspondências entre os dois grafos de dependência recai no problema conhecido como Isomorfismo entre Grafos[49].

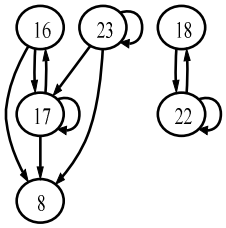
Baseado nestes conceitos, esta tese apresenta o uso de uma metodologia de identificação de códigos executáveis metamórficos de origem maliciosa, através da conversão destes códigos para linguagem *assembly* e posterior construção dos grafos de dependência correspondentes, que serão então comparados com uma base de referência para sua identificação como *malware*. A Figura 3.1 apresenta um exemplo de grafo de dependência gerado a partir de um código *assembly*.

```
linha:01 .686p
linha:02 .model flat
linha:03    push eax
linha:04    push ebx
linha:05    push ecx
linha:06    call sub_01
linha:07 ini_loop:
linha:08    cmp ebx, eax
linha:09    jg  end_loop
linha:10    call sub_02
linha:11    jmp ini_loop
linha:12 end_loop:
linha:13    call sub_03
linha:14    end
linha:15 sub_01 proc near
linha:16    mov eax, 9
linha:17    mov ebx, 3
linha:18    mov ecx, 0
linha:19 retn
linha:20 sub_01 endp
linha:21 sub_02 proc near
linha:22     add ecx, 1
linha:23     sub eax, ebx
linha:24 retn
linha:25 sub_02 endp
linha:26 sub_03 proc near
linha:27    pop ecx
linha:28    pop ebx
linha:29    pop eax
linha:30 retn
linha:31 sub_03 endp
```

a) Código assembly



b) Grafo de dependência original



c) Grafo de dependência reduzido

Figura 3.1 - Exemplo de um código assembly (a) e seus grafos de dependência original (b) e reduzido (c), construídos a partir das dependências do código semântico.

3.2. Metodologia de Identificação Utilizada

O processo de identificação de códigos executáveis metamórficos, exibido na Figura 3.2, é composto por quatro etapas principais:

1. **Reconstrução de código *assembly***, onde o programa executável passa por um processo de engenharia reversa para a obtenção do seu equivalente em linguagem *assembly*.
2. **Geração do grafo de dependência**, onde o programa gerado na etapa anterior é analisado então usado como base de geração do grafo de dependência.
3. **Redução do grafo**, onde o grafo de dependência, obtido na etapa anterior, passa

por um processo que elimina vértices e arestas oriundos principalmente do processo de metamorfismo do código. Além disso, partes do código onde o controle de fluxo nunca irá passar também são removidas. De acordo com nossa proposta, um tratamento adicional de redução deve ser executado para os grafos que constituírem a base de referência.

4. **Comparação do grafo reduzido com a base de referência**, onde o grafo reduzido na etapa anterior é comparado com um grafo correspondente a um *malware* previamente analisado.

Mais detalhes sobre cada etapa da metodologia de identificação são apresentados nas seções seguintes, incluindo as estruturas de dados e ferramentas desenvolvidas para a construção de um protótipo utilizado para os testes de avaliação da metodologia.

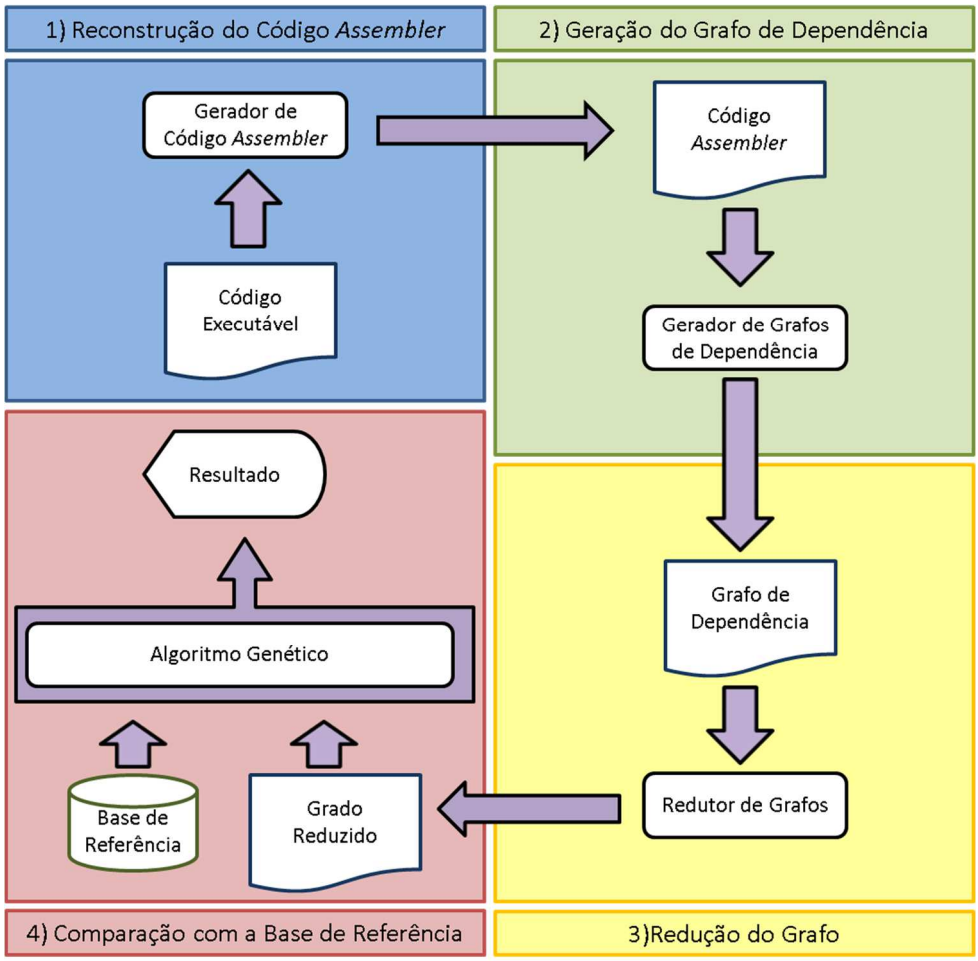
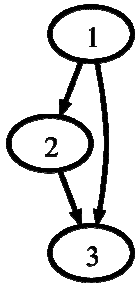


Figura 3.2 - Metodologia utilizada para identificação de códigos maliciosos metamórficos.

3.3. Representação de Grafos de Dependência

De forma a garantir uma comunicação eficiente entre as interfaces de entrada e saída de cada componente, foi utilizado um padrão simples de representação e codificação das informações referentes a um grafo direcionado, ilustrado na Figura 3.3.

O grafo de dependência é representado como uma sequência de números inteiros, codificados em forma não textual. O primeiro número representa a quantidade total de vértices presentes no grafo e os demais números representam as arestas, onde o primeiro número do par indica o vértice de origem e o segundo o vértice de destino. Este modelo pode ser formalmente definido como: Seja $n \in N^*$, onde n é quantidade de vértices de um grafo direcionado $G = (V, E)$, onde V é o conjunto de vértices definidos por $V = \{i \mid 1 \leq i \leq n\}$ e E é o conjunto de arestas direcionadas definidas por $E = \{(i, j) \mid i, j \in V\}$. Dessa forma, o grafo apresentado na Figura 3.3.a pode ser codificado pela sequência numérica 3, 1, 2, 1, 3, 2 e 3 (Figura 3.3.b).



a) Grafo original

```
3, 1,
2, 1,
3, 2,
3
```

b) Sequência de codificação

Figura 3.3 - Exemplo do padrão de representação de um grafo de dependência.

3.4. Reconstrução do Código Assembler

O processo identificação de códigos maliciosos metamórficos a partir de um programa executável começa com a execução de um processo de engenharia reversa, que transforma o código binário executável em um programa escrito em linguagem *assembly*. Várias ferramentas, também conhecida como *disassemblers*, podem ser usadas nesse processo como OllyDbg [50] e IDA Pro [51].

Infelizmente, este processo de reversão não é perfeito. Algumas limitações podem ser identificadas no momento que os códigos gerados são analisados detalhadamente. Além disso, o fato dos códigos maliciosos terem sofrido a ação de

técnicas de metamorfismo adiciona ainda outras limitações, conforme exemplificado na Figura 3.4. Os problemas mais comuns encontrados nos códigos reconstruídos são: i) instruções de desvio e chamadas a funções com base no conteúdo de registradores ou variáveis e; ii) trechos de código armazenados na área de dados.

Estes problemas introduzem componentes dinâmicos que não podem ser determinados no momento da construção do grafo de dependência, pois o conteúdo desses elementos só poderia ser determinado durante a execução do código, o que foge do escopo do processo de análise estática utilizada neste trabalho. No caso dos trechos

```
CODE:00401273 push    ecx
CODE:00401274 mov     ecx, 20007Eh
CODE:00401279 call    near ptr unk_40337C
CODE:0040127E mov     ecx, eax

CODE:004028C0 call    near ptr byte_402EDF
CODE:004028C5 jz      loc_4028D0
CODE:004028CB jmp     near ptr byte_402EF1

CODE:00402792 loc_402792:
CODE:00402792 mov     cl, 6
CODE:00402794 jmp     near ptr dword_402EFC
```

a) Desvios e chamadas de funções dinâmicas

```
DATA:00403016
DATA:00403016 loc_403016:
DATA:00403016 push    es
DATA:00403017 jnz     near ptr unk_403565
DATA:0040301D mov     al, 0F6h
DATA:0040301F add     al, 95h
DATA:00403022 mov     [edi], al
DATA:00403024 add     edi, 1
DATA:00403027 push    ecx
```

b) Código armazenado na área de dados

Figura 3.4 - Exemplos de trechos de código que geram diferentes componentes conexos e que não podem ser tratados diretamente pelo processo de geração de Grafos de Dependência.

de código armazenados na área de dados, estes poderiam ser modelados como componentes conexos distintos do grafo principal. Entretanto, como normalmente estes trechos não são constituídos de muitas instruções, poderiam ser relacionados com outras seções de código não associadas ao código malicioso. Assim, este trabalho optou por ignorar estes elementos dinâmicos.

3.5. Geração dos Grafos de Dependência

O código *assembly* gerado na fase anterior é submetido a um processo que mapeia as relações de dependência entre instruções, registradores e variáveis que cada instrução manipula. Cada instrução é associada com um vértice e a manipulação dos registradores/variáveis definirá quais arestas serão criadas.

O processo de geração das arestas inicia com a identificação dos registradores/variáveis manipulados em cada instrução. Para cada um desses elementos de armazenamento de dados, uma das seguintes ações pode ser se tomada:

i. Caso o elemento esteja sendo manipulado pela primeira vez, o vértice correspondente àquela instrução é marcado como origem para futuras manipulações daquele mesmo elemento;
ii. Se o elemento já tenha uma origem definida, é criada uma nova aresta direcionada no grafo partindo da origem e tendo como destino o vértice correspondente à instrução atual, e;
iii. Quando a instrução estiver alterando o conteúdo do elemento, além da criação de uma nova aresta, a origem é atualizada para o vértice correspondente à instrução atual.

Este processo ainda prevê a necessidade de reavaliação de instruções, em função da presença de instruções de desvio de fluxo do programa. Isto ocorre porque a origem pode ter sido alterada dentro de um laço ou chamada de procedimento, o que cria novas relações de dependência que devem ser analisadas. Entretanto, o procedimento de reavaliação deve ser executado com algum cuidado, principalmente no caso das instruções de desvio condicional, visto que a linguagem *assembly* não possui instruções de controle de fluxo (por exemplo, o `if-then-else`) ou instruções de laço, pois todos os controles são implementados através de instruções de desvio do tipo "*jump*". Assim, as relações de dependência mapeadas devem ser compatíveis com o fato de que:

a) Trechos de código podem ser ignorados, o que abre a possibilidade do estabelecimento de relações de dependência entre as instruções localizadas antes de depois do trecho ignorado. Este tipo de situação ocorre, por exemplo, quando do uso de instruções `if-then-else`;
b) Trechos de código podem ser executados mais de uma vez, o que pode gerar relações de dependência entre instruções posicionadas em porções de código anteriores ao da instrução atual, além de relações de dependência de uma instrução para ela mesma (um laço).

Assim, para cada desvio condicional presente, abrem-se dois possíveis caminhos alternativos para o fluxo de execução. Os caminhos alternativos podem então ser modelados como uma árvore binária cheia e completa com $\mathbf{2^n}$ folhas, onde $\boldsymbol{n}$ representa

a quantidade de instruções de desvios condicionais presentes no código.

Caso todos os caminhos desta árvore fossem percorridos, o tempo gasto tornaria todo o processo impraticável. Para tratar este problema, foi utilizada uma lista que armazena os valores correspondentes aos vértices de origem presentes no momento que cada instrução de desvio vai ser avaliada. Desta forma, quando aquela mesma instrução for avaliada novamente, uma consulta a esta lista é realizada e, caso os vértices de origem presentes naquele momento sejam iguais a um dos estados previamente salvos, o procedimento de reavaliação é interrompido, em um processo de poda que salva tempo de processamento e torna a extração de grafos de dependência viável.

Além disso, para evitar que o processo de geração entre num laço infinito, seja pela presença de instruções de desvio para endereços menores que o da instrução atual ou pela presença de chamadas recursivas de procedimentos, foi utilizada uma estrutura de pilha que é consultada para que instruções já analisadas sejam ignoradas. Isto permite o avanço na avaliação do código e a conclusão do processo de geração de grafos.

3.6. Redução dos Grafos de Dependência

Reduzir um grafo de dependência significa eliminar os vértices e arestas associadas que são considerados desnecessários, tais como vértices que representam a declaração de variáveis e vértices que representam trechos do código que nunca serão atingidos e que dão origem a componentes conexos distintos no grafo. Ao fim do processo de redução obtém-se um grafo que representa o comportamento principal do código, como o exemplo mostrado na Figura 3.5, que representa o resultado da redução aplicada ao grafo mostrado na Figura 3.1.b.

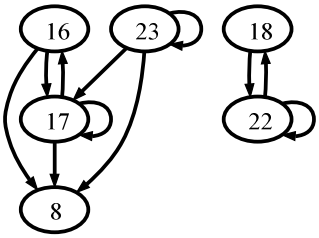


Figura 3.5 - Grafo de dependência reduzido.

Para realizar a redução dos grafos de dependência foram definidas quatro situações onde os vértices devem ser eliminados [15]:

1) Vértices com apenas uma aresta de saída e sem arestas de entrada;
2) Vértices com apenas uma aresta de entrada e sem arestas de saída;

3) Vértices com apenas uma aresta de entrada e uma aresta de saída; e
4) Vértices que não possuem nenhuma aresta de entrada ou saída.

Após estes ajustes os grafos gerados estão prontos para serem utilizados na etapa seguinte.

3.7. Identificando Elementos Relevantes do Grafo

Como já citado anteriormente, o processo de comparação entre grafos é um problema NP-Difícil. Entretanto, se for possível diminuir o tamanho dos grafos que estão sendo comparados, sem que esta redução elimine as caraterísticas que permitem distingui-lo dentro de um conjunto de grafos de mesma natureza, este processo de comparação pode ser otimizado de forma significativa.

Os grafos de dependência derivados de programas executáveis possuem características intrínsecas que permitem a identificação dos componentes mais relevantes do grafo, tornando possível uma redução aprimorada do grafo, o que contribui significativamente na melhoria dos resultados no processo de comparação.

3.7.1. Diferenciação dos Vértices em Grafos de Dependência

Em função de análises estruturais dos vértices pertencentes à grafos de dependência, extraídos de programas executáveis, e da correlação destes vértices com os as instruções presentes nos códigos *assembly,* usados para gerar estes grafos, esta tese propõe classificar estes vértices em três categorias distintas: a) vértices de partida; b) vértices de processamento; e c) vértices de decisão. Estes tipos de vértices são ilustrados na Figura 3.6.

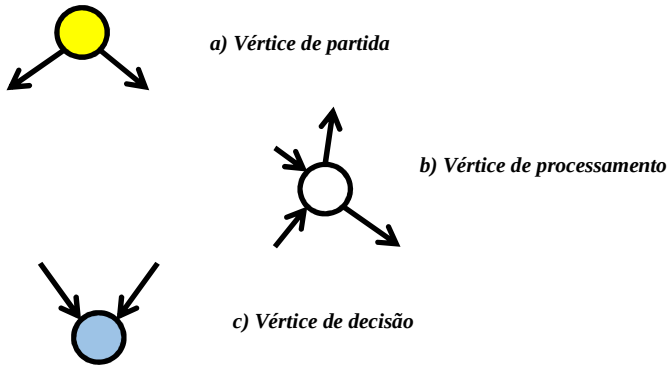


Figura 3.6 - Tipos de vértice encontrados em grafos de dependência.

Os **Vértices de Partida** dizem respeito ao início da cadeia de dependências associadas à manipulação de uma variável ou registrador. Como estas instruções correspondem no código à primeira manipulação na cadeia de dependências, os vértices derivados não possuem arestas incidentes, assim como uma raiz em uma árvore de busca. Entretanto, diferente de árvores de busca tradicionais, os grafos de dependência possuem mais de um nó com este tipo de características. Este tipo de nó também serve como a indicação de pontos de partida, a partir dos quais o programa original poderia iniciar o seu processamento. Na Figura 3.7, os nós 1, 13 e 56 são exemplos deste tipo de vértice.

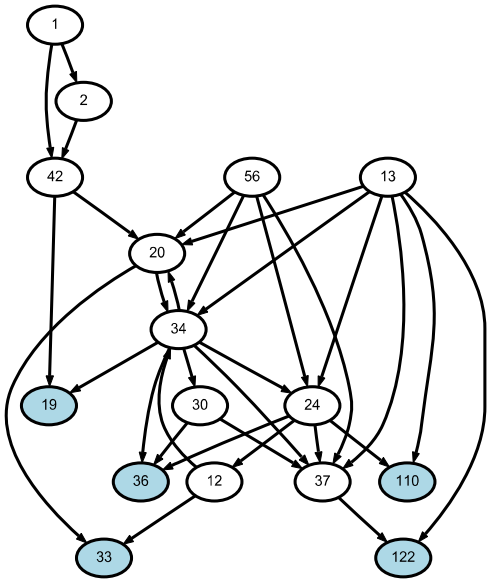


Figura 3.7 - Grafo de dependência reduzido, com os vértices de decisão em destaque.

Os **Vértices de Processamento** estão associados a qualquer instrução que manipulam e alteram o conteúdo dos registradores e variáveis presentes no código. A característica estrutural que identifica este tipo de vértice é a presença tanto de arestas incidentes como arestas partindo destes nós. A maioria dos vértices presentes nos grafos de dependência, gerados a partir dos códigos manipulados neste trabalho, fazem parte desta categoria. Na Figura 3.7, alguns exemplos deste tipo de vértice incluem os nós 2, 20, 34 e 42.

Finalmente, os **Vértices de Decisão** são aqueles gerados a partir das instruções `CMP`, utilizadas para avaliar e comparar o conteúdo de registradores e variáveis. Estas instruções podem ser encontradas antes de qualquer instrução de desvio condicional é o seu resultado que realmente determina se o desvio será ou não executado. Um exemplo deste tipo de instrução pode ser encontrado na linha 8 do código *assembly* apresentado

na Figura 3.1.a. com o vértice correspondente podendo ser identificado no grafo de dependência reduzido que é apresentado na Figura 3.5. Outros exemplos deste tipo de vértice incluem os nós 19, 33, 36, 110 e 122 na Figura 3.7.

Como programas escritos em *assembly* não possuem instruções de controle de alto nível como `if-then-else` e `do-while`, a instrução `CMP` é a principal ferramenta responsável pela implementação das estruturas de tomada de decisão e do controle do fluxo do programa. Em outras palavras, é através destas instruções que a maior parte da lógica de funcionamento do programa é construída. Como estas instruções não alteram o conteúdo dos registradores e variáveis avaliadas, os vértices gerados a partir delas têm a característica distinta de não possuírem arestas de saída, como as folhas em uma árvore binária de pesquisa.

Como as arestas representam as relações de dependência entre as instruções, podemos supor que quanto maior for a quantidade de arestas incidindo num vértice de decisão, maior será sua importância para a implementação da lógica básica do programa modelado. É com base nestes vértices que o processo de redução aprimorado que propomos neste trabalho é capaz de decidir que outros elementos podem ser descartados.

É comum também encontrar mais de um componente conexo nestes grafos de dependência, derivados de elementos de controle não tão relevantes para o processo de identificação. Por exemplo, na Figura 3.5, o subgrafo formado pelos vértices 18 e 22 é derivado da manipulação de um contador, podendo ser desconsiderado no processo de identificação que, a partir daí, se concentraria apenas no restante do grafo. A Figura 3.8 ilustra um grafo de dependência gerado a partir de um *malware* real, onde podem ser identificados alguns componentes conexos bem menores o que o componente principal da estrutura do grafo.

Para identificar os vértices de decisão mais relevantes para o programa modelado, foi utilizado um processo dividido em 4 etapas: 1) cálculo da menor distância relativa entre cada vértice de decisão; 2) construção de um grafo virtual derivado, constituído apenas dos vértices de decisão, com arestas representando a distância relativa entre cada vértice de decisão; 3) cálculo da clique máxima [52] presente neste grafo virtual derivado; e 4) redução final do grafo de dependência, com a eliminação de qualquer vértice e aresta que não estejam associados aos vértices de

decisão presentes na clique máxima do grafo virtual derivado. As próximas seções apresentam este processo com mais detalhes.

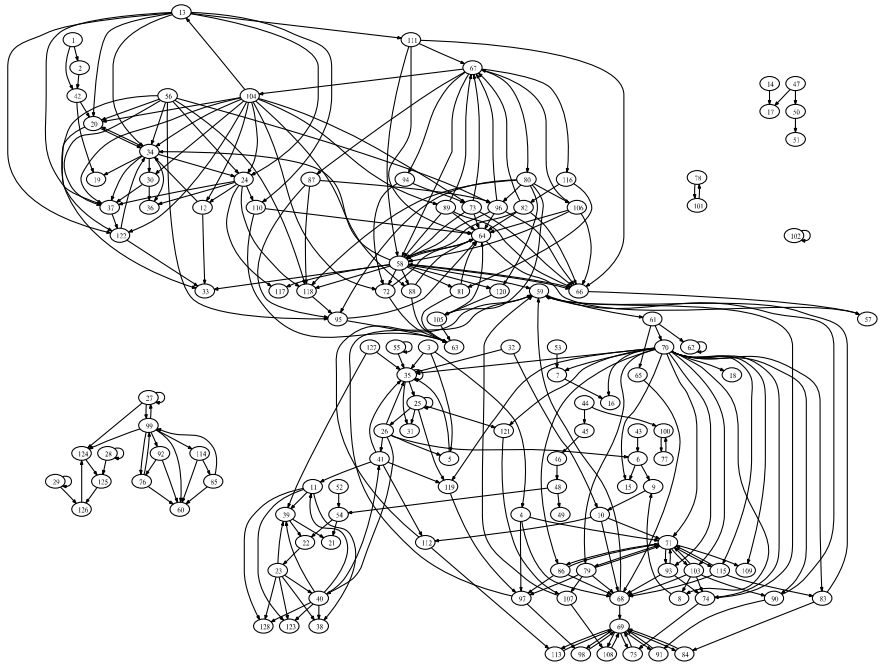


Figura 3.8 - Grafo de dependência gerado com base no *malware* W32.Evol.

3.7.2. Cálculo da Menor Distância Relativa Entre Vértices de Decisão

Tradicionalmente, a utilização de um algoritmo como Floyd-Warshall [46][47] poderia ser empregado para a obtenção da distância entre cada vértice de um grafo. Entretanto, como os vértices de decisão, por definição, não possuem arestas que se originam a partir deles, a distância calculada a partir destes vértices para qualquer outro vértice presente no grafo de dependência, sempre seria infinita.

Assim, para efeitos desta metodologia, no momento do cálculo da distância entre cada vértice de decisão, é considerada a existência do conjunto de arestas virtuais que invertem o sentido das arestas originalmente incidentes nos vértices de decisão, criando uma conexão de saída que permite o cálculo da distância relativa entre estes vértices. A Figura 3.9 ilustra as arestas virtuais (destacadas em vermelho) criadas para o grafo apresentado na Figura 3.7.

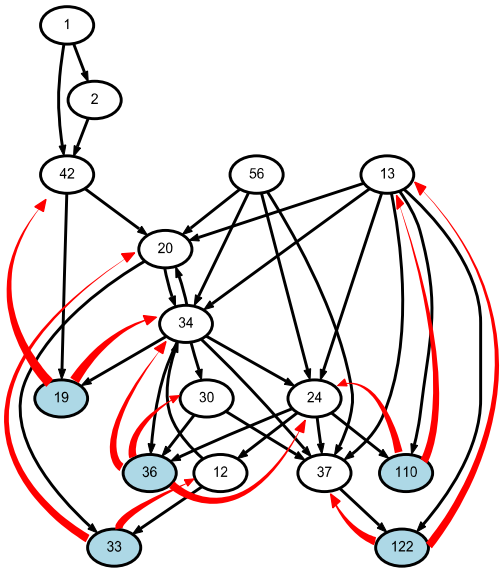


Figura 3.9 - Arestas virtuais adicionadas ao grafo de dependência da Figura 14.

Para que estas arestas virtuais não interfiram no cálculo das distâncias mínimas para os outros nós, foi desenvolvida uma adaptação do algoritmo de Floyd-Warshall com duas alterações fundamentais: 1) todos os vértices de decisão não são considerados como elementos intermediários para o cálculo das distâncias mínimas entre os vértices; e 2) quando os vértices de origem são identificados como "de decisão" as arestas virtuais são consideradas no cálculo das distâncias mínimas entre este vértice e todos os demais. Um resultado que merece destaque é que, em função da primeira alteração, este novo algoritmo, apesar de continuar intrinsecamente cúbico em sua ordem de complexidade, na prática não executará todas as n^3 interações, já que os dois loops mais internos não serão executados quando o vértice intermediário considerado for de decisão. O algoritmo modificado é apresentado na Figura 3.10. Assim, com todas as distâncias calculadas, a próxima etapa do processo pode ser executada.

Algorithm 1 Floyd-Warshall Modificado

```
1: for each v_k in G
2:   if v_k isn't a decision vertex
3:     for each v_o
4:       for each v_d
5:         if v_o isn't a decision vertex
6:           if (d(v_o,v_k) + d(v_k,v_d)) < d(v_o,v_d)
8:             set d(v_o,v_d)=(d(v_o,v_k) + d(v_k,v_d))
9:         else
10:          if (d(v_k,v_o) + d(v_k,v_d)) < d(v_o,v_d)
11:            set d(v_o,v_d)=(d(v_k,v_o) + d(v_k,v_d))
```

Figura 3.10 - Algoritmo de Floyd-Warshall modificado, com a inclusão das linhas 2 e 5 para tratamento diferenciado quando o vértice manipulado for do tipo de decisão.

3.7.3. Construção do Grafo Virtual Derivado

Com todas as distâncias mínimas calculadas, a etapa seguinte cria um grafo virtual onde cada vértice corresponderá a um dos vértices de decisão do grafo de dependência original e as arestas serão geradas com base na distância calculada entre cada um destes vértices. Caso este grafo virtual fosse gerado com base no grafo apresentado na Figura 3.9, a matriz de adjacências apresentada na Figura 3.11 seria produzida. É interessante destacar que as distâncias calculadas entre dois vértices nem sempre são as mesmas, dependendo do sentido da aresta.

	19	**33**	**36**	**110**	**122**
19	2	3	2	3	3
33	3	2	3	4	5
36	2	3	2	2	3
110	3	3	2	2	2
122	3	3	3	2	2

Figura 3.11 - Matriz de adjacências correspondente ao grafo virtual.

3.7.4. Cálculo do Clique Máximo no Grafo Virtual.

Este processo visa identificar quais os vértices de decisão são os mais relevantes para o funcionamento do programa que está sendo modelado. Para esta etapa foi aplicada uma metodologia tradicional de identificação do clique máximo [55] ao grafo virtual gerado na etapa anterior. No grafo representado pela matriz de adjacências da Figura 3.7, o clique virtual é apresentado na Figura 3.12.

Como consequência, vértices que corresponderem a elementos desconexos do grafo e vértices que tiverem baixa conectividade com os demais serão desconsiderados.

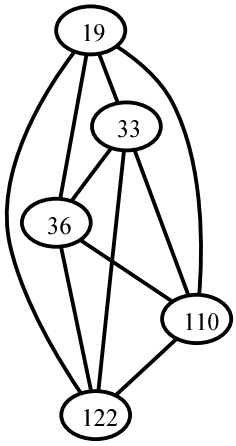


Figura 3.12 - Clique para o grafo virtual na Figura 18.

3.7.5. Redução Final do Grafo de Dependência

Nesta etapa do processo de redução final do grafo de dependência, a lista de vértices pertencentes ao clique virtual é usada para determinar se os vértices e arestas presentes no grafo de dependência deverão ou não permanecer na versão final do grafo de dependência reduzido.

Para continuar fazendo parte da versão final do grafo de dependência reduzido, o vértice deve possuir um caminho no grafo reduzido original que o ligue até um dos vértices presentes no clique do gravo virtual. Caso ele não possua este caminho, este vértice e todas as arestas associadas devem ser eliminados. O resultado deste processo é ilustrado na Figura 3.13, que apresenta a versão final do grafo de dependência reduzido do *malware* W32.Evol. Esta nova versão reduzida que será usada para identificação de versões metamórficas deste mesmo *malware*.

3.8. Comparação dos Grafos de Dependência

Os algoritmos de comparação de grafos buscam encontrar uma solução através da construção iterativa de associações estabelecidas entre os vértices de dois grafos, G_1 e G_2, que satisfaça um conjunto de restrições do problema em análise. Neste trabalho, o algoritmo de comparação de grafos tem como objetivo gerar uma solução viável para o problema de isomorfismo de grafos.

Neste contexto, a diferenciação entre os tipos de vértices, presentes nos grafos de dependência, é introduzida de forma a aprimorar o processo de comparação e medição do nível de similaridade entre estes grafos. Nesta visão, o conjunto de vértices

V em cada grafo tratado é constituído por dois grupos distintos: a) V_d , o subconjunto de *V* formado apenas por vértices de decisão e b) V_p , o subconjunto de *V* formado apenas pelos vértices de processamento.

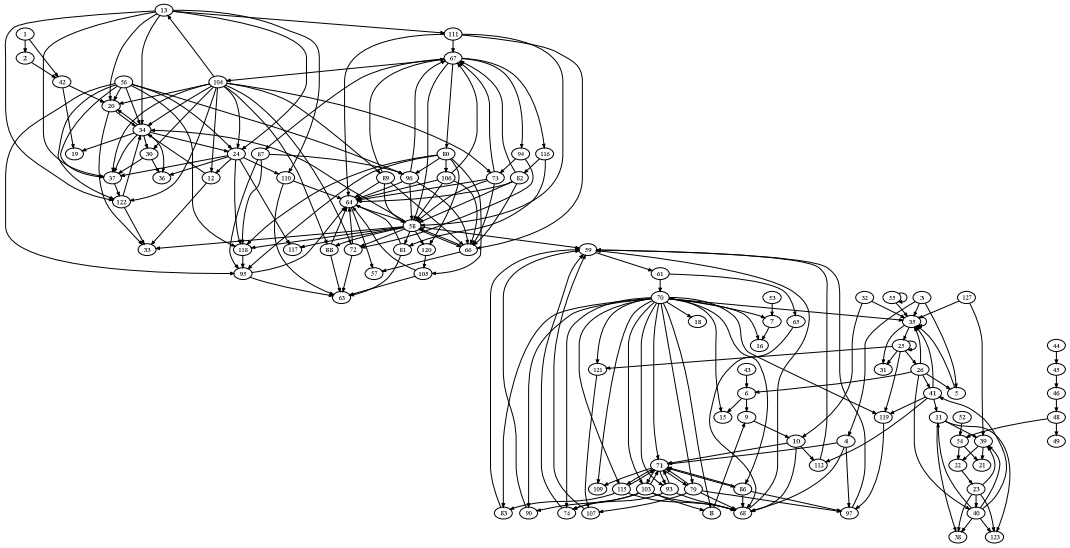


Figura 3.13 - Versão final do grafo de dependência reduzido do W32.Evol.

Podemos então afirmar que, no escopo desde trabalho, seja *G=(V, E)* um grafo de dependência gerado a partir de um programa executável, o conjunto de vértices *V* pode ser representado de acordo com a Equação (1).

$$V=V_d, \cup\ V_p \qquad (1)$$

Esta diferenciação contribui para a precisão dos resultados porque o processo de comparação usa esta informação para evitar que vértices de natureza distinta sejam comparado entre si.

É importante destacar que, neste processo de comparação, nem todos os vértices (sejam de decisão ou processamento) contribuem para pontuação de similaridade, já que a quantidade de vértices de decisão e de processamento comparados deve ser iguais. Por exemplo, caso G_1 tenha 10 vértices de decisão, 25 de processamento e G_2 tenha 9 vértices de decisão e 50 vértices de processamento, os subgrafos comparados (G_1' e G_2') terão apenas 9 vértices de decisão e 25 vértices de processamento.

Em função destas características, todo o subgrafo $G_i'=(V_i', E_i')$ de G_i que será objeto da comparação, obedece às propriedades representadas na Equação (2).

$$G'=(V', E') \mid G' \subset G, V' \subset V, E' \subset E \qquad (2)$$

Assim, para que a quantidade de vértices em cada subgrafo analisado sejam

iguais, o processo de comparação verifica o tamanho do conjunto de vértices de dependência de G_1 e G_2 e identifica qual destes possui o menor tamanho, de forma a definir este tamanho como a quantidade máxima de vértices de decisão que cada subgrafo comparado deve possuir. Este mesmo processo é aplicado com relação aos vértices de processamento presentes em G_1 e G_2, de forma a que os subgrafos G_1' e G_2' tenham a mesma quantidade de vértices de decisão e vértices de processamento, garantindo que as restrições descritas na Equação (3) sejam obedecidas.

$$G_1' \subset G_1 \, , \, G_2' \subset G_2 \mid |V_{d1}'| = |V_{d2}'|, \, |V_{p1}'| = |V_{p2}'| \qquad (3)$$

Com a geração dos subgrafos, o processo de comparação está pronto para iniciar a etapa de medição do nível de similaridade presente entre estes dois grafos de dependência. A pontuação de similaridade faz uso de uma função *I(e,E)* que verifica se uma determinada aresta *e* pertence a um conjunto de arestas *E*. Se a aresta for encontrada no conjunto, a função retorna o valor 1 (um) e caso a aresta não seja encontrada, o valor retornado será 0 (zero), conforme descrito na Equação (4).

$$I(e, E) = \begin{cases} 1, & se \quad e \in E \\ 0, & caso \quad contrário \end{cases} \qquad (4)$$

Para se utilizar a função *I* iniciamos com a extração dos conjuntos de arestas E_1' e E_2', a partir dos subgragos G_1' e G_2'. A partir deste ponto, para cada vértice de G_1' são testadas as arestas $e \in E_1'$ tal que *e* é uma aresta que se origina a partir desde vértice. Esta aresta é testada em relação ao conjunto de arestas de G_2', de forma que se em E_2' existir uma aresta idêntica, a função $I(e, E_2')$ retorna o valor 1. A seguir, o processo é repetido com relação aos vértices e arestas de G_2', com relação ao conjunto de arestas de G_1'.

A esta pontuação inicial é acrescentada ainda uma pontuação extra derivada dos conjuntos de arestas virtuais Λ_1' e Λ_2' formados a partir das as arestas virtuais geradas a partir dos vértices de decisão presentes nos subgrafos G_1' e G_2'. Assim de forma análoga aos conjuntos de teste anteriores, λ é uma aresta virtual tal $\lambda \in \Lambda_1'$, que se origina a partir de um vértice de decisão pertencente a G_1'. Se este vértice tiver um equivalente em Λ_2', a função $I(\lambda, \Lambda_2')$ retorna o valor 1 aumentando a pontuação de similaridade. Novamente, o processo se repete para as arestas $\lambda \in \Lambda_2'$ de G_2' que são testadas em relação ao conjunto Λ_1' de G_1', fechando a pontuação inicial de similaridade.

Finalmente, de forma a que a pontuação de similaridade possa ser apresentada de uma forma discreta e de fácil interpretação, esta pontuação inicial é dividida pela soma do tamanho dos conjuntos E_1', E_2', Λ_1' e Λ_2', normalizando a pontuação de similaridade.

Isto dá origem à função de cálculo de similaridade entre grafos de dependência gerados a partir de programas executáveis, expressa na Equação (5), que calcula a pontuação de similaridade final usada para determinar se existe ou não contaminação por *malware* no programa analisado.

$$similaridade(G_1', G_2') = \frac{\sum_{e \in E_1'} I(e, E_2') + \sum_{e \in E_2'} I(e, E_1') + \sum_{\lambda \in \Lambda_1'} I(\lambda, \Lambda_2') + \sum_{\lambda \in \Lambda_2'} I(\lambda, \Lambda_1')}{|E_1'| + |E_2'| + |\Lambda_1'| + |\Lambda_2'|} \quad (5)$$

As pontuações geradas a partir do uso desta função de similaridade variam de 0 (zero), o que indica que os grafos comparados não possuem nenhuma aresta em comum, até 1 (um), o que indica que estes grafos são idênticos (100% de similaridade), o que fornece uma representação clara do nível de similaridade presente entre os dois subgrafos comparados.

3.9. Características do Algoritmo Genético de Comparação.

O processo base de comparação das amostras utiliza um algoritmo genético, que deixa o subgrafo G_1' inalterado e gera diferentes arranjos de vértices do grafo G_2' que, a partir daí, são comparadas com o grafo G_1' através do uso da função de similaridade detalhada na Equação (5).

Cada um desses arranjos representa um membro de uma população de *n* elementos. Esta população inicial é renovada a medida que o processo de comparação avança, onde *k* novos filhos (definidos pela taxa de substituição) substituem os *k* arranjos com menor pontuação existentes na população. Este processo se repete até que seja atingido um limite máximo de gerações definidos pelos parâmetros de controle do algoritmo genético.

Finalmente, tirando proveito da classificação dos vértices, o processo de geração da população pode ainda ser configurado em dois modos de operação distintos: 1) comparação simples por tipo de vértice; e 2) comparação com restrição de escopo.

3.10. Introduzindo a Diferenciação de Vértices na Comparação de Grafos de Dependência.

A metodologia de Kim e Moon [15] compara os grafos de dependência através da execução de um algoritmo genético que avalia a similaridade entre pares de vértices, comparando as arestas que cada vértice possui, buscando sempre uma melhora no resultado a cada iteração.

Entretanto, durante a execução deste processo de comparação, como o modelo original não faz qualquer tipo de distinção entre a natureza de cada vértice, o algoritmo genético original consome parte de seu tempo comparando vértices de processamento com vértices de decisão, desperdiçando tempo de processamento desnecessariamente. Este trabalho pretende eliminar este desperdício, através do uso ativo da classificação de vértices para que elementos de natureza distinta não sejam pareados, durante o processo de comparação, sem que para isto seja necessário alterar o modelo de codificação utilizado para armazenar os grafos na base de referência.

Isto é possível mesmo que as informações de classificação não sejam salvas na estrutura de armazenamento dos grafos de dependência, uma vez que o custo de obtenção desta informação é absorvido pelo processo de leitura destas estruturas, no momento da recuperação das informações necessárias para criar as arestas pertencentes ao grafo de dependência original. Afinal, todo e qualquer vértice posicionado como origem de uma aresta será automaticamente reconhecido como vértice de processamento. Todo e qualquer vértice que não apresentar esta característica, será então classificado como vértice de decisão. Este processo classificará os vértices de partida como vértices de processamento, mas isto é esperado, pois vértices de partida podem se transformar em vértices de processamento, e vice-versa, em função da alteração na ordem que as instruções são inseridas no código, como ação do metamorfismo.

Assim, para tirar proveito da classificação dos vértices, este trabalho propõe e avalia a utilização de duas abordagens de comparação distintas: 1) comparação segmentada por tipo de vértice; e 2) comparação segmentada com restrição de escopo.

Entretanto, antes de iniciar qualquer uma destas abordagens, é necessário que os

vértices pertencentes aos grafos que serão comparados sejam separados em dois subconjuntos distintos. No primeiro conjunto, ficam apenas os nós de decisão e no segundo conjunto, os nós de processamento e nós de partida. Os nós de partida não receberam tratamento diferenciado, pois sua quantidade é muito menor e, dada a possibilidade de reposicionamento de instruções, uma mesma instrução pode tanto gerar um nó de partida como um nó de processamento. Com os dois conjuntos separados, podemos dar início ao processo de comparação.

3.10.1. Comparação Segmentada por Tipo de Vértice

Na abordagem de comparação segmentada por tipo de vértice (ou comparação por tipo), empregamos as mesmas estratégias genéticas utilizadas na abordagem de referência. O diferencial aqui é que são executados dois processos de comparação em paralelo, onde os conjuntos gerados na etapa anterior são comparados de acordo com o seu tipo. Assim apenas vértices que possuam natureza similar serão comparados entre si, otimizando o processo de comparação como um todo.

Outra característica particular desta abordagem é que, como os vértices de decisão estão em menor quantidade, o processo de comparação destes conjuntos pode ser executado em uma quantidade menor de ciclos, terminando antes e deixando o restante do tempo de processamento dedicado apenas à comparação de nós de processamento e partida. A Figura 3.14 ilustra a diferença entre a disposição de cromossomos sem o uso da segmentação (Figura 3.14.a) e com o uso da segmentação (Figura 3.14.b), para o grafo de dependência da Figura 3.7.

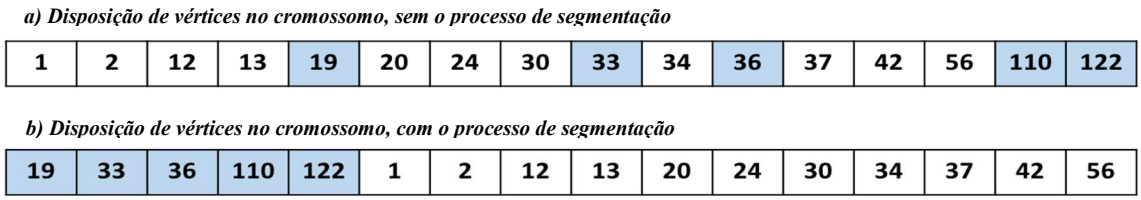


Figura 3.14 – Comparação entre a disposição de vértices em cromossomos sem segmentação e com o uso de segmentação.

3.10.2. Comparação Segmentada com Restrição de Escopo

Na comparação segmentada com restrição de escopo (ou comparação por escopo), também ocorre uma comparação diferenciada pelo tipo de vértice. Entretanto, os

vértices de processamento comparados serão apenas aqueles que podem ser atingidos a partir dos vértices de decisão selecionados para o primeiro grupo de comparações. Estes resultados são obtidos através da manipulação da população inicial que é repassada ao algoritmo genético para comparação.

No momento da construção da população inicial, os vértices de processamento serão dispostos de acordo com a ligação que possuírem com os vértices de decisão, de forma a que fiquem organizados de acordo com a distância que se encontraram destes vértices.

Este processo de construção da população inicial é dividido em 4 etapas:

1) Os vértices de decisão são inseridos nas primeiras posições do cromossomo. No primeiro membro da população estes vértices são posicionados em ordem crescente, baseado em seu rótulo, nos demais elementos estes vértices de decisão são dispostos aleatoriamente;
2) A lista de vértices de processamento ligados diretamente a cada um dos vértices de decisão é recuperada. Estes vértices são os primeiros vértices de processamento a serem inseridos no cromossomo. A ordem na qual estas listas são recuperadas obedece à mesma disposição dos vértices de decisão contidos no cromossomo que está sendo construído. Entretanto, mesmo que um vértice pertença a mais de uma lista, ele será inserido apenas na sua primeira ocorrência. Este processo continua até que todos os vértices de processamento ligados diretamente a vértices de decisão sejam incluídos;
3) Agora é a vez dos primeiros vértices de processamento inseridos terem suas listas de vértices adjacentes consultadas. Assim, para cada um dos vértices de processamento já inseridos é recuperada a lista de vértices atingidos A partir dele. Estes vértices são então inseridos no cromossomo, desde que já não tenham sido inseridos anteriormente. Este processo se repete até que todos os vértices de processamento presentes no cromossomo sejam avaliados, tenham eles sido inseridos na etapa anterior ou na etapa atual;
4) A última etapa se encarrega de inserir todos os vértices que não tenham sido inseridos nas etapas anteriores. Este pequeno conjunto de vértices diz respeito a dois tipos de vértices: a) vértices de partida, que não podem ser atingidos pelo processo de construção do cromossomo descrito nas etapas anteriores, pois estes vértices só possuem arestas originadas a partir deles, não existindo qualquer caminho no grafo que leve até eles; e b) vértices de

processamento que só podem ser atingidos a partir dos vértices de partida.

Todo este processo organiza o os vértices de uma maneira que é relacionada à organização topológica do grafo original o que proporciona melhores resultados no processo de comparação entre os cromossomos. Além disso, como a maioria dos vértices de processamento está próxima dos vértices de decisão do primeiro grupo, isto diminui o número de elementos a serem trabalhados e otimiza o tempo de processamento, diminuindo o número de iterações necessárias para gerar uma pontuação próxima da máxima pontuação de similaridade entre os grafos comparados.

Outra possibilidade proporcionada por este método é a delimitação de uma distância máxima que os vértices de processamento precisam ter em relação aos vértices de decisão escolhidos para comparação, limitando ainda mais o número de elementos que serão manejados durante o processo de levantamento do nível de similaridade entre grafos. A Figura 3.15 ilustra a diferença de disposição entre o processo de segmentação (Figura 3.15.a) e o processo de restrição de escopo (Figura 3.15.b), para o grafo da Figura 3.7.

Os resultados obtidos com as duas metodologias demonstram que a segmentação com restrição de escopo tem resultados mais fiéis ao real nível de similaridade entre os grafos de dependência. Em função disto, a comparação restrição de escopo é a metodologia empregada na versão final do processo de identificação.

a) Disposição de vértices no cromossomo, com o processo de segmentação

19	33	36	110	122	1	2	12	13	20	24	30	34	37	42	56

b) Disposição de vértices no cromossomo, com o processo restrição de escopo

19	33	36	110	122	34	42	12	20	24	30	13	37	1	2	56

Figura 3.15 – Comparação entre a disposição de vértices em cromossomos com segmentação e com restrição de escopo.

Agora que toda a metodologia de identificação proposta foi apresentada, o próximo capítulo apresenta os resultados obtidos através da utilização da mesma, bem como apresenta uma descrição detalhada de como os resultados de pontuação de similaridade são interpretados durante o processo de identificação.

Capítulo 4

Resultados

Este Capítulo apresenta o protocolo experimental aplicado para avaliar a metodologia proposta, detalhando os experimentos e discutindo os resultados obtidos pela introdução da diferenciação de vértices do processo de redução aprimorado e na identificação do nível de similaridade entre grafos de dependência.

4.1. Protocolo Experimental

Para ter um controle maior do desempenho da metodologia proposta por esta tese, o processo de testes e avaliação da metodologia de identificação de *malware* metamórfico foi dividido em duas etapas distintas: i) testes de unidade e ii) teste de integração [56]. A Figura 4.1 apresenta uma visão geral dos componentes do protocolo experimental.

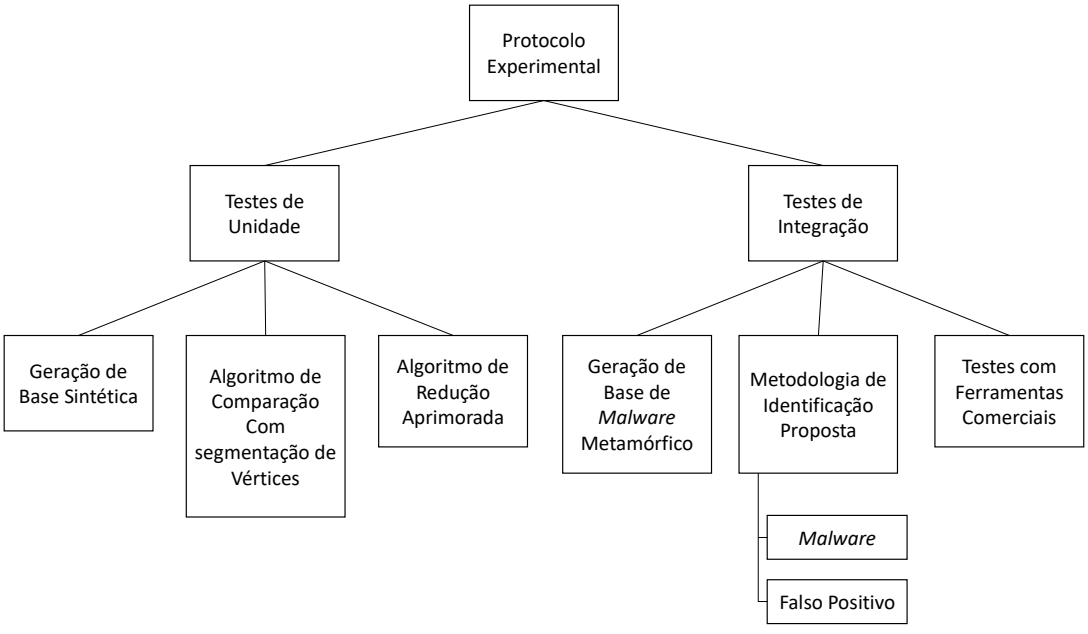


Figura 4.1 - Protocolo experimental para avaliação da metodologia proposta.

Durante os testes de unidade, os módulos responsáveis por implementar as funcionalidades de comparação e redução de grafos de dependência são testados individualmente, para determinar se o seu funcionamento está de acordo com o

esperado, além de determinar os padrões de operacionais para os testes que foram executados na etapa seguinte.

Esta primeira etapa de testes se inicia com a geração de uma base de grafos sintética, onde todos os grafos são gerados de acordo com um perfil predefinido, facilitando o processo de interpretação dos resultados obtidos pelo algoritmo de comparação, cujas funcionalidades foram desenvolvidas de acordo com as características listadas na seção 3.9. Como o nível de similaridade das amostras é conhecido previamente, características como estabilidade e precisão podem ser avaliadas de forma precisa.

Os testes com o processo de redução aprimorado utilizam dois conjuntos de grafos de dependência gerados a partir de dois *malware* metamórficos distintos obtidos. Inicialmente, os grafos de referência de cada conjunto são submetidos ao processo de redução aprimorado. Neste momento é possível medir qual foi o nível de redução obtido em cada um dos grafos.

A seguir, fazendo uso do algoritmo de comparação selecionado (seção 3.9.2), os conjuntos de amostras são comparados tanto com seu grafo de referência original, como com o grafo de referência reduzido, para que novamente seja avaliado tanto a estabilidade e como a precisão do processo de identificação baseado no grafo de referência reduzido, quando comparado com os resultados obtidos a partir do grafo de referência original.

De posse dos resultados obtidos na etapa anterior, a etapa de testes de integração emprega, de forma conjunta, o módulo de redução e o módulo comparação, na tentativa avaliar tanto a identificação de amostras metamórficas reais como os resultados a partir de um conjunto de amostras não contaminadas.

Esta etapa inicia com a definição do nível de pontuação obtido quando o grafo de referência reduzido é comparado com o grafo de referência original. Esta pontuação é utilizada para calcular o limiar mínimo que o processo de comparação baseado no grafo de dependência reduzido deve atingir para que uma identificação seja indicada como positiva.

Com o valor do limiar mínimo para identificação de cada um dos grupos de *malware* definidos, o próximo passo envolve a geração de 100 (cem) amostras metamórficas de *malware*, geradas através do uso de uma ferramenta de mutação

externa. Nos primeiros testes desta fase, o grupo de amostras metamórficas é comparado com o grafo de referência reduzido para avaliação da eficiência do processo de identificação proposto nesta tese. Nesse teste, quanto maior for a quantidade de amostras identificadas com sucesso, melhor é a avaliação do desempenho do algoritmo de comparação. Em seguida, para avaliação de como a metodologia proposta lida com falsos positivos, os grafos de referência reduzidos são comparados tanto com grupos de grafos de dependência pertencentes a outras famílias de *malware*, como com um grupo de grafos de dependência gerados a partir de programas livres de qualquer tipo de contaminação. Neste caso, quanto menor for a taxa de identificação, melhor será o desempenho do algoritmo.

Finalmente, algumas das versões metamórficas geradas serão submetidas à avaliação de ferramentas de antivírus profissionais, gerando uma referência a partir do qual o desempenho da metodologia proposta nesta tese pode ser comparado.

As próximas seções detalham como cada uma das etapas descritas neste protocolo experimental foi executada, além de apresentar os resultados de todos os estes desenvolvidos.

4.2. Testes de Unidade

O primeiro grupo de testes foi utilizado para validar as características individuais dos dois componentes que empregaram o conceito de diferenciação de vértices: o comparador e o redutor aprimorado. Entretanto, antes de iniciar estes testes, foi necessário criar uma base de dados sintética, para que os resultados obtidos pudessem ser comparados diretamente com as características projetadas dos grafos gerados. As próximas seções detalham estes experimentos.

4.2.1. Geração da Base de Grafos Sintética

Para a realização dos testes sintéticos foi criada uma base de grafos gerados a partir de uma metodologia usada na avaliação de algoritmos para detecção de subgrafos[57]. Tal metodologia propõe as seguintes regras:

i. Os vértices nunca terão arestas que apontam para si mesmos, não possuindo laços ou *loops*;
ii. Deve-se manter um grau de similaridade entre o par, chamado de MCS —

Maximum Common Subgraph ou maior subgrafo comum —, isso significa que os grafos do par devem possuir uma quantidade mínima de vértices e arestas iguais;

iii. Os grafos devem possuir somente um componente conexo.

Para determinar a quantidade de arestas que um grafo deve possuir, usamos a equação $|E|=p.n.(n-1)$, onde p representa o valor da probabilidade de conexão entre os vértices e n indica a quantidade de vértices do grafo, tal que $|V|=n$. Assim, quanto maior for o valor de *p*, maior será a quantidade de arestas do grafo.

Neste trabalho serão gerados 10 pares para cada combinação de parâmetros, o que resultou na geração e comparação de 2.520 instâncias de grafos. A Tabela 4.1 apresenta a descrição e os valores utilizados de cada parâmetro.

Tabela 4.1 - Parâmetros utilizados para gerar os grafos utilizados no teste.

Parâmetro	Significado	Valores
MCS	Porcentagem de similaridade entre os grafos do par	10%, 30%, 50%, 70% e 90%
p	Probabilidade de conexão entre dois vértices	1%, 5%, 10% e 20%
n	Quantidade de vértices dos grafos	10, 15, 20, 25, 30, 35, 40, 50, 60, 70, 80, 90 e 100

4.2.2. Testes de Comparação da Base de Grafos Sintéticos

Para realizar os testes de comparação de grafos, foram utilizados os grafos pertencentes ao mesmo par, pois estes possuem uma relação de similaridade conhecida (valor *MCS*), o que possibilita a avaliação da eficiência do algoritmo de comparação utilizado.

Para avaliar a diferença de desempenho entre as técnicas de comparação por tipo e comparação por escopo, a base de dados sintética foi submetida à avaliação destas duas técnicas. A Figura 4.2 apresenta dois gráficos de dispersão, obtidos a partir de cada uma das técnicas. Estes gráficos relacionam os valores das pontuações de diferenças, obtidos pelo cálculo de similaridade descrito na seção 3.9, com os valores do MCS e a quantidade de vértices.

A partir da comparação dos resultados obtidos por cada uma das metodologias, é possível perceber que os resultados da comparação sem restrição de escopo (Figura 4.2.a) possuem um comportamento similar e estável na maioria dos experimentos. Entretanto, as pontuações obtidas foram, na maioria das vezes, bem superiores aos resultados esperados. Este efeito foi principalmente observado nas faixas onde o MCS

possuía o menor valor. Apenas as faixas de valor de MCS superiores à 70% atingiram pontuações mais condizentes com as características apresentadas pelos grafos analisados. Assim, apesar das faixas de MCS estarem todas separadas por um fator linear (cada faixa possui um valor de MSC foi gerada em saltos de 20%), as faixas de pontuação obtidas tiveram um comportamento não-linear, quando relacionadas umas às outras

Nos resultados obtidos com a utilização da comparação com restrição de escopo (Figura 4.2.b), os níveis de pontuação obtidos pelas amostras com menor quantidade de vértices, foram similares àqueles obtidos na comparação por tipo. Entretanto, à medida que a quantidade de vértices aumentou, o nível de pontuação gerado foi se comportando cada vez mais de acordo com aquele compatível com as características das amostras analisadas. Assim, para as amostras com uma quantidade de vértices maior, as faixas de pontuação geradas tiveram um comportamento mais linear quando comparadas umas com as outras, apesar dos resultados não serem tão estáveis quantos àqueles obtidos nos experimentos com a comparação por tipo.

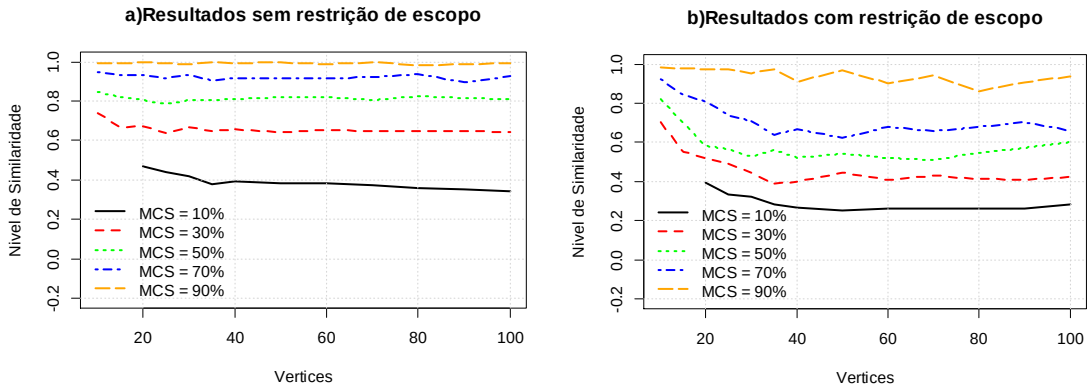


Figura 4.2 - Gráfico de dispersão relacionando as médias das diferenças com os valores do MCS e da quantidade de vértices, obtidos através das duas metodologias de comparação.

Em função destes resultados, a metodologia de comparação por escopo foi definida como a metodologia padrão a ser empregada em todos os demais experimentos.

Através destes experimentos foi possível avaliar o desempenho das duas metodologias de comparação proposta. Entretanto, este experimento apresentou uma limitação. Como todas as comparações empregaram pares de grafos com a mesma quantidade de arestas, os níveis de pontuação gerados não foram os mesmos obtidos quando se comparou os grafos de dependência reduzidos, utilizados como base de

referência para as comparações com *malware*, com outros grafos que representavam instâncias metamórficas destes *malware*.

Assim, foi necessário estabelecer um nível de pontuação de referência que levasse em consideração a diferença na quantidade de vértices e à redução final que o grafo de referência é submetido. Este processo está descrito na seção 4.3.2.

4.2.3. Resultados do Processo de Redução Aprimorado

Para construir a base de grafos de referência para o processo de identificação, o conjunto de arquivos utilizados como base para a geração das 100 amostras metamórficas, foi submetido a todo o processo de extração de grafos de dependência.

Entretanto, diferente das amostras metamórficas, este conjunto de grafos foi submetido ao processo de redução aprimorado, conforme descrito na seção 3.7.5. Os resultados deste processo são apresentados na Tabela 4.2.

Tabela 4.2 – Percentual de redução da quantidade de vértices nos grafos de referência para o processo de comparação.

Malware	**Redução Original**	**Redução Aprimorada**	**Percentual de Redução**
Conficker	61	42	31,15%
Klez.A	163	120	26,38%
Sircam.A	32	28	12,50%
StormWorm	24	23	4,17%
Stuxnet	325	184	43,38%

O percentual de redução variou desde 4,17% até 43,38%. Através da análise dos grafos originais, percebeu-se que o processo de redução produziu um maior impacto nas amostras constituídas por grafos com mais de um componente conexo. Entretanto, mesmo quando o grafo possuía apenas um componente conexo, o processo de redução aprimorado conseguiu identificar elementos menos relevantes que poderiam ser eliminados do grafo de referência final.

Em uma etapa anterior deste projeto de pesquisa[58], foi determinado que esta redução aprimorada torna o processo de comparação mais estável, além de melhorar a identificação de amostras metamórficas. Em testes realizados com amostras dos *malware* W32.Evol e W32.Polip, a pontuação obtida apresentou um coeficiente de

variação (desvio padrão dividido pela média) menor quando o grafo de referência utilizado foi aquele gerado pelo processo de redução aprimorado. Além disto, nos testes com o W32.Polip, amostras que não tinham sido identificadas originalmente, passaram a obter um resultado de identificação positiva, através do uso do grafo com redução aprimorada.

Os resultados destes experimentos são ilustrados na Figura 4.3, e apresentados de forma resumida na Tabela 4.3.

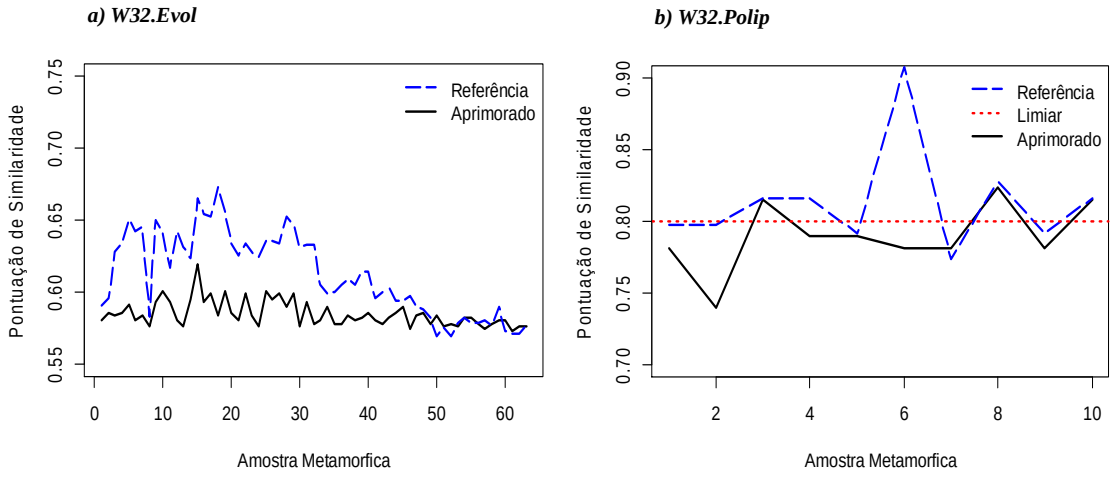


Figura 4.3 - Comparação entre os resultados obtidos com o uso de um grafo reduzido apenas pelo processo tradicional e àqueles obtidos pelo processo de redução aprimorado, para os códigos maliciosos W32.Evol (a) e W32.Polip (b).

Tabela 4.3 –Resultados de testes com o W32.Evol e o W32.Polip.

	Coeficiente de Variação (%)		Taxa de Sucesso (%)	
Malware	**Original**	**Aprimorado**	**Original**	**Aprimorado**
W32.Evol	4,65	1,51	100	100
W32.Polip	4,51	3,05	50	70

Outro resultado importante deste processo de redução aprimorado é a obtenção de dois grafos relacionados ao mesmo *malware* (os grafos obtidos antes e depois do processo de redução aprimorado) que são usados para definir os níveis mínimos de pontuação de similaridade usados para a identificação positiva de uma amostra metamórfica de *malware*. Este processo é descrito em detalhes na seção 4.5.

4.3. Testes de Integração

Os testes de integração combinam os resultados dos componentes independentes, o que permite avaliar toda a metodologia de identificação de malware metamórficos. Novamente, o primeiro passo desta etapa é a geração da base de grafos para teste. Entretanto, aqui foram empregados grafos de dependência gerados a partir de *malware* reais, submetidos a um processo de metamorfismo externo. As próximas seções detalham estes experimentos.

4.3.1. Geração da Base de *Malware* Metamórficos

Para a realização dos testes de integração foi criada uma base de malware metamórficos gerados a partir de amostras não metamórficas dos *malware* Conficker, Klez.A, Sircam.A, StormWorm e Stuxnet. Originalmente, outros *malware* também tinham sido selecionados para este teste. Entretanto, por limitações das ferramentas de engenharia reversa, não foi possível extrair com sucesso os grafos de dependência destas amostras.

A seguir, cada uma das amostras não metamórficas foi usada como base para a geração de outras 100 (cem) amostras metamórficas, através do uso da ferramenta *Code Pervertor*[1]. Cada nova amostra era gerada com base na amostra anterior. As assinaturas MD5 de cada arquivo gerado foram verificadas para garantir que todas as amostras possuíam algum tipo de diferença que distinguisse uma das outras.

Ao final deste processo, todas as amostras que fizeram parte dos testes eram diferentes umas das outras. O arquivo original foi utilizado para a geração do grafo de referência e as amostras metamórficas foram a base para os grafos de dependência que foram alvo do processo de comparação. Dessa forma, o material base para testes estava pronto para a próxima fase.

4.3.2. Testes com a Metodologia de Identificação Proposta

Quando os testes com as amostras metamórficas foram realizados, as pontuações obtidas em cada grupo geraram resultados de pontuação bem distintos. Ao serem comparadas com o modelo de referência gerado a partir da base sintética, em três dos cinco grupos as pontuações obtidas foram muito abaixo de 20% de similaridade. Este mesmo comportamento tinha sido previamente observado em uma etapa anterior deste

[1] http://vxheaven.org/vx.php?id=tc04

projeto de pesquisa [58]. Apesar disso, cada grupo comparado apresentou níveis de pontuação muito similares quando analisados isoladamente.

Assim, para se obter uma visão mais clara do nível de pontuação que seria adequado para cada amostra, foi executado um experimento onde o grafo de referência foi comparado com sua versão anterior ao processo de redução final. Dessa forma era garantido que o grafo de referência seria um subgrafo do grafo comparado. Cada uma das amostras foi submetida a este processo em ciclos de 100 repetições. Os resultados deste experimento são apresentados na Figura 4.4.

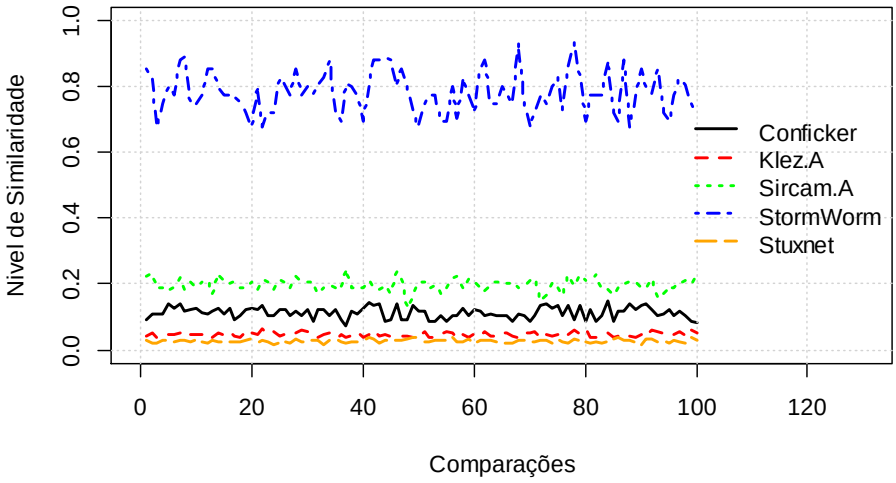


Figura 4.4 – Resultados da comparação dos grafos de referência com suas versões não reduzidas.

Estes resultados demonstraram que cada grupo de amostras possuía um nível de pontuação diferente, relacionado diretamente ao grafo e ao *malware* usado como referência de comparação. Assim, as pontuações obtidas neste experimento definiram as bases de pontuação que cada grupo de amostras metamórficas deveria atingir para ser considerada como positiva a identificação de um programa analisado. Além disso, foi também definido que todas as amostras metamórficas seriam submetidas a ciclos de dez comparações, com a pontuação de cada amostra sendo definida pela média desses resultados.

Finalmente, para que a diferença na quantidade de vértices também fosse levada em consideração no momento da geração da pontuação final, foi aplicado um coeficiente de ajuste da pontuação, gerado a partir da quantidade de vértices de cada grafo sendo comparado (grafo de referência *vs* grafo do programa analisado), onde o menor valor é dividido pelo maior, gerando um redutor de pontuação. Este redutor é

aplicado à pontuação original através da multiplicação destes dois valores, obtendo-se assim o valor final da pontuação de similaridade de cada teste. Um exemplo dos resultados desse processo é apresentado na Figura 4.5, onde é apresentada a pontuação de similaridade final das versões metamórficas geradas a partir do *malware* Conficker. Como limite mínimo de detecção, ou o **limiar de identificação**, foi definida uma pontuação 10% inferior ao nível de pontuação de similaridade de referência. A taxa de identificação neste experimento foi de 98% das amostras. Os resultados obtidos em todos os testes de identificação são apresentados de forma resumida na Tabela 4.4.

Além destes testes, também foram avaliadas as ocorrências de identificações de falsos positivos. A Figura 4.6 apresenta os resultados do uso do grafo de referência do *malware* Conficker aplicado na identificação das amostras metamórficas do Klez.A.

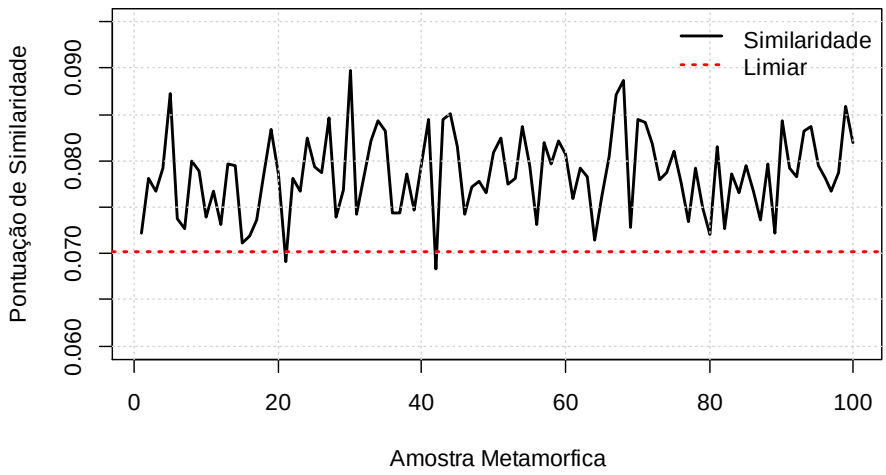


Figura 4.5 - Utilizando o coeficiente de redução na pontuação de similaridade de referência para Conficker metamórfico.

Tabela 4.4 – Resumo dos resultados de testes de identificação de versões metamórficas de *malware*.

***Malware* Base**	**Vértices no Grafo de Referência**	**Média de Vértices nas Amostras Metamórficas**	**Taxa de Identificações com Sucesso**
Conficker	42	60,51	98%
Klez.A	120	99,97	100%
Sircam.A	28	33	99%
StormWorm	23	24	100%
Stuxnet	184	332,93	97%

Os resultados apresentados na Figura 4.6 demonstram que a pontuação obtida foi muito abaixo do nível mínimo estipulado, não ocorrendo nenhuma identificação errônea. Um resumo de todos os testes de falso positivos realizados é apresentado na Tabela 4.5.

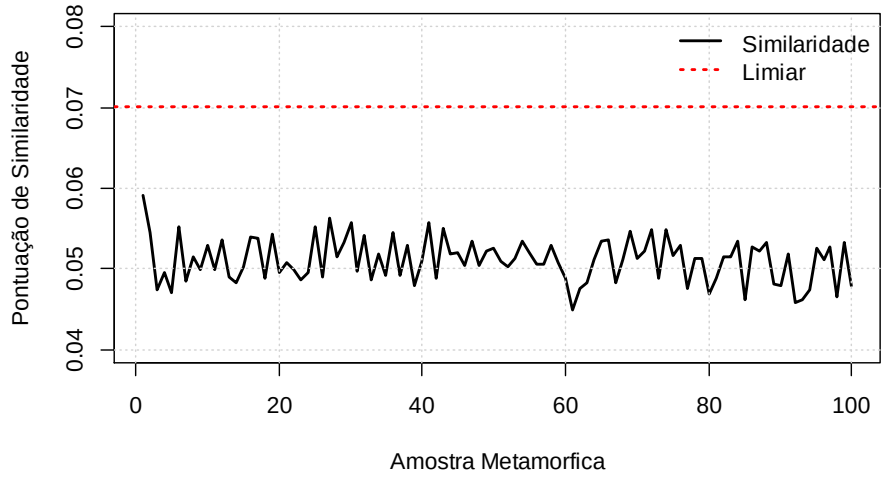


Figura 4.6 – Teste de falso positivo usando como referência o grafo do Conficker nas amostras metamórficas do Klez.A.

Tabela 4.5 – Resumo dos resultados de testes de falsos positivos.

Malware **Base**	**Amostras**	**Referência**	**Média de Vértices nas Amostras**	**Taxa de identificações Errôneas**
Conficker	Klez.A	42	99,97	0%
Conficker	Stuxnet	42	332,93	0%
Sircam.A	Klez.A	28	99,97	0%
Sircam.A	Stuxnet	28	332,93	0%
StormWorm	Conficker	23	60,51	0%
StormWorm	Klez.A	23	99,97	0%
StormWorm	Sircam.A	23	33	0%
StormWorm	Stuxnet	23	332,93	0%

Um último teste de falso positivo foi realizado usando como referência o grafo do *malware* Sircan.A comparado com grafos de dependência extraídos de 28 programas idôneos e livres de qualquer contaminação. Novamente, cada amostra foi comparada diversas vezes com o grafo de referência e tomada como pontuação final a média dos resultados. A Figura 4.7 apresenta os resultados deste teste. A taxa de falsos positivos gerada foi de 7%.

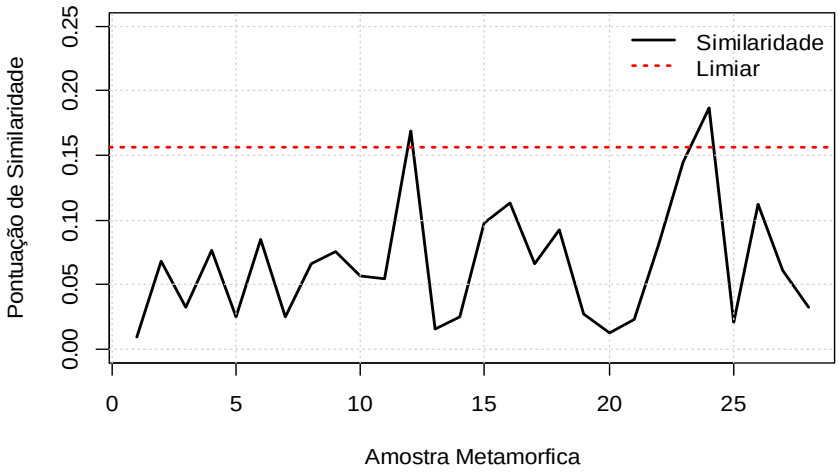


Figura 4.7 – Teste de falso positivo usando como referência o grafo do Sircam.A em grafos de dependência de programas livres de contaminação.

4.3.3. Testes com Ferramentas Comerciais

Para concluir os testes de unidade, algumas das amostras metamórficas geradas para estes experimentos foram submetidas à ferramenta Vírus Total[2]. Nos testes realizados, esta ferramenta submeteu cada arquivo a uma quantidade entre 55 e 57 ferramentas de antivírus comerciais, para que cada uma delas possa tentar identificar o arquivo submetido. Um exemplo do resultado deste teste é apresentado na Tabela 4.6.

Apesar das ferramentas comerciais implementarem diferentes estratégias para tratar o problema do *malware* metamórfico, elas ainda estão longe de serem soluções efetivas. Nos testes apresentados na Tabela 4.6, apenas 27 das ferramentas comerciais foram capazes de identificar a presença de um *malware* e destas, apenas 16 identificaram corretamente o tipo de *malware*. Em média 65,3% dos testes obtiveram uma identificação positiva da presença de código malicioso. Entretanto, apenas 18,2% das amostras metamórficas foram identificadas corretamente, quanto ao tipo de *malware* a partir do qual se originaram, indicando que muito esforço ainda é necessário para que uma proteção dessas ferramentas seja completa. A Tabela 4.7 apresenta um resumo dos resultados de todos os testes realizados no Vírus Total.

[2] https://www.virustotal.com/pt/

Tabela 4.6 – Resultados dos testes com uma amostra metamórfica do Klez.A avaliado por 55 ferramentas comerciais.

Nome do arquivo:	*Klez.A_pvtd_3.exe*
Taxa de detecção:	*27 / 55*
Ferramenta	***Malware Identificado***
ALYac	Generic.Klez.CF2012EF
AVG	Win32/ElKern
AhnLab-V3	Worm/Win32.Klez
Arcabit	Generic.Klez.CFD7DCEF
Avast	Win32:Vitro
BitDefender	Generic.Klez.CF2012EF
ByteHero	Virus.Win32.Part.f
CAT-QuickHeal	(Suspicious) - DNAScan
Cyren	W32/Heuristic-257!Eldorado
DrWeb	Win32.HLLM.Klez.origin
ESET-NOD32	probably unknown NewHeur_PE
Emsisoft	Generic.Klez.CF2012EF (B)
F-Prot	W32/Heuristic-257!Eldorado
F-Secure	Generic.Klez.CF2012EF
GData	Generic.Klez.CF2012EF
Ikarus	Virus.Win32.Elkern
Jiangmin	I-Worm/Klez.b
Kaspersky	Virus.Win32.Elkern.b
McAfee	W32/Klez.worm.gen
McAfee-GW-Edition	BehavesLike.Win32.Klez.km
MicroWorld-eScan	Generic.Klez.CF2012EF
Microsoft	Virus:Win32/Elkern.A.dll
Qihoo-360	QVM19.1.Malware.Gen
Symantec	W32.Klez.A@mm
TheHacker	W32/Klez.gen@MM
VBA32	Win32.Klez.3326
nProtect	Generic.Klez.CF2012EF

Tabela 4.7 – Resumo dos testes com a ferramenta Vírus Total.

Malware									
Conficker		**Klez.A**		**Sircam.A**		**StormWorm**		**Stuxnet**	
Identificações	**Testes**	**Identificações**	**Testes**	**Identificações**	**Testes**	**Identificações**	**Testes**	**Identificações**	**Testes**
37	56	33	55	37	56	38	55	40	56
36	55	31	55	37	56	40	55	41	56
34	55	28	53	35	56	38	56	38	55
37	55	27	55	36	55	40	56	37	55
35	55	32	55	36	56	37	55	39	56
35	53	29	55	36	54	42	55	36	55
35	55	28	52	37	56	39	56	40	55
34	55	33	56	38	56	40	56	36	56
36	55	32	54	39	56	37	56	41	55
36	55	29	56	37	56	42	55	37	56
35,5	**54,9**	**30,2**	**54,6**	**36,8**	**55,7**	**39,3**	**55,5**	**38,5**	**55,5**
Médias									

Um resultado que demonstra como a base de dados de ferramentas comerciais pode possuir diversas entradas para versões metamórficas de um mesmo *malware* é apresentado na Tabela 4.8. Nesta tabela, apresentamos as diferentes identificações que algumas ferramentas de antivírus comerciais apresentaram para amostras metamórficas do malware *Confiker*. Em média 71% das ferramentas que foram capazes de identificar a amostra submetida como um *malware*, apresentaram a uma única identificação para todas as variações metamórficas. Para as demais ferramentas os resultados variaram entre 2 e 5 identificações com rótulos distintos.

Tabela 4.8 – Exemplos de identificações distintas para o mesmo *malware*, com base na ferramenta Vírus Total.

Nome do Malware:	*Conficker*
Ferramenta	***Malware Identificado***
AegisLab	Net.Worm.W32!c
AegisLab	W32.W.Kido.li1O
AegisLab	W32.W.Kido.lDiQ
ClamAV	Trojan.Dropper-18535
ClamAV	Win.Dropper.Agent-35454
Cyren	W32/Trojan.ERHK-1795
Cyren	W32/Trojan.KNSI-5907
Cyren	W32/Trojan.MKCB-1822
Cyren	W32/S-f436afa4!Eldorado
McAfee	W32/Conficker.worm
McAfee	Artemis!726A55D27469
McAfee	Artemis!1D71530646D0
McAfee	Artemis!79CE772A65B3
McAfee	Artemis!B3E815D9201E
NANO-Antivirus	Trojan.Win32.Kido.buctx
NANO-Antivirus	Trojan.Win32.Kido.dyssbb
Panda	W32/Conficker.C.worm
Panda	Generic Suspicious
Qihoo-360	Win32/Worm.a79
Qihoo-360	Malware.Radar01.Gen
Qihoo-360	QVM31.1.Malware.Gen
Tencent	Trojan.Win32.Qudamah.Gen.13
Tencent	Win32.Worm-net.Kido.Akpq
Tencent	Win32.Worm-net.Kido.Fib
Zillya	Adware.BrowseFox.Win32.174170
Zillya	Worm.Kido.Win32.12
Zillya	Worm.Kido.Win32.2672
Zillya	Trojan.Agent.Win32.592309

Finalmente, devemos reconhecer que, apesar das ferramentas comerciais terem apresentado resultados variados, os testes mostraram que a maioria das ferramentas mais utilizadas e conhecidas foram eficiente em identificar a contaminação das amostras, mesmo que tenham falhado em apresentar um resultado único.

4.4. Considerações Finais sobre os Testes

Os resultados obtidos através destes testes demonstram que a metodologia proposta foi capaz de identificar a maioria dos códigos maliciosos metamórficos que foram avaliados. Algumas ferramentas comerciais obtiveram resultados semelhantes, apesar de mais de 30% das ferramentas testadas não terem sido capazes de identificar estas amostras e mesmo estas identificações não apontaram a identidade correta em mais de 80% dos testes. No caso da metodologia proposta nesta tese, todas as identificações de *malware* indicaram corretamente a identidade do código malicioso original.

Capítulo 5

Conclusões e Trabalhos Futuros

A diferenciação entre os tipos de vértices presentes nos grafos de dependência tratados nesta pesquisa ofereceu contribuições tanto para o processo de redução aprimorado dos grafos de referência, como para o processo de comparação e identificação do nível de similaridade entre estes grafos.

A identificação dos elementos estruturais mais relevantes, baseada no uso da clique virtual, para o mapeamento dos vértices de decisão que estão mais interacionados, possibilitou a diminuição dos grafos de referência e o aprimoramento dos resultados tanto de forma qualitativa, com a diminuição do coeficiente de variação, como quantitativa, com o aumento da taxa de identificações com sucesso.

Quanto às contribuições ao processo de comparação, é importante lembrar que a metodologia de referência[15], não faz qualquer tipo de distinção entre a natureza de cada vértice, o que deixa o algoritmo genético gastar parte de seu tempo na tentativa de comparar vértices de processamento com vértices de decisão, desperdiçando tempo de processamento de forma desnecessária.

Este trabalho apresentou uma forma de eliminar este desperdício, através do uso ativo da classificação de vértices para que elementos de natureza distinta não sejam pareados, durante o processo de comparação, sem que para isto fosse necessário alterar o modelo de codificação utilizado para armazenar os grafos na base de referência.

Isto é possível mesmo que as informações de classificação não sejam salvas na estrutura de armazenamento dos grafos de dependência, uma vez que o custo de obtenção desta informação é absorvido pelo processo de leitura destas estruturas, no momento da recuperação das informações necessárias para criar as arestas pertencentes ao grafo de dependência original. Afinal, todo e qualquer vértice posicionado como origem de uma aresta será automaticamente reconhecido como vértice de processamento. Qualquer vértice que não apresentar esta característica, será então classificado como vértice de decisão. Este processo classificará os vértices de partida

como vértices de processamento, mas isto é esperado e não introduz problemas ao processo de comparação.

Finalmente, a comparação segmentada por escopo introduziu um aprimoramento significativo aos resultados do processo de medição do nível de similaridade entre os grafos de dependência manipulados neste trabalho, tornando os resultados mais coerentes com os verdadeiros níveis de similaridade presentes nas estruturas comparadas.

5.1. Limitações da Metodologia Proposta

Na metodologia proposta neste trabalho, a primeira limitação está associada ao processo de engenharia reversa dos códigos executáveis, pois a fidelidade dos grafos de dependência gerados depende muito do nível de sucesso na geração dos códigos *assembly* derivados dos programas executáveis sob análise. Infelizmente, em muitos casos os códigos gerados não possuíam informações suficientes para a extração de grafos de dependência, seja pela presença de blocos de instrução de carga dinâmica, ou pela simples incapacidade da ferramenta de traduzir corretamente os códigos executáveis para instruções em linguagem *assembly*. Um processo engenharia reversa dinâmico, que envolvesse também a análise do código em execução, poderia solucionar parte destes problemas, mas o restante do processo ainda dependeria diretamente da eficiência da fermenta de extração de códigos *assembly*.

Outra limitação relevante está associada ao desempenho do algoritmo de comparação utilizado. Apesar de ter sido possível estabelecer os níveis mínimos de pontuação de similaridade, necessários para a identificação de cada grupo de malware, através da comparação do grafo de referência com redução aprimorada com sua versão original, na verdade a pontuação esperada nestes experimentos seria de 100% de similaridade, já que temos garantido que o grafo com redução aprimorada é uma versão reduzida do grafo original.

Como o algoritmo de comparação teve desempenho adequado no grupo de grafos sintéticos, onde todos os experimentos de comparação usavam pares de grafos com a mesma quantidade de vértices, as limitações apresentadas nos testes com grafos extraídos de *malware* reais, onde todas as comparações lidavam com grafos que possuíam quantidades de vértices diferentes, parece estar diretamente relacionada com esta diferença na quantidade de vértices. Assim, é fundamental encontra abordagens

alternativas para o processo de comparação.

Finalmente, uma limitação inerente ao processo de redução é a possibilidade da criação de forma acidental de falsos vértices de decisão. Caso todas as arestas que se originam de um certo vértice de processamento se destinarem a vértices que forem removidos durante o processo de redução inicial ou mesmo na redução aprimorada, em função das suas novas características estruturais, este vértice passará a ser tratado pela metodologia como um vértice de decisão, o que certamente impactará negativamente em sua avaliação de similaridade. Infelizmente não foi possível avaliar este cenário nos testes que foram executados. Uma modificação na forma como as informações dos grafos de dependência são armazenadas pode solucionar este problema, mas isto exigiria um ajuste em todas as ferramentas desenvolvidas, o que inviabilizou este procedimento para o conjunto de testes que foram executados nesta tese.

Apesar destas limitações, os resultados demonstram que todas as técnicas propostas contribuem de forma bastante efetiva na identificação de códigos maliciosos metamórficos, permitindo que os objetivos propostos neste trabalho fossem plenamente atingidos.

5.1. Trabalhos Futuros

A partir da análise das limitações destacadas na seção 5.1, é possível destacar alguns caminhos que podem evoluir os resultados obtidos até aqui.

A inclusão da análise dinâmica do código a ser investigado, para minimizar os problemas relacionados aos componentes dinâmicos e a trechos de código não reconstruídos deve aprimorar a fidelidade dos grafos de dependência relacionados. O uso de um ambiente de *sandbox* é fundamental para este fim.

O emprego de metodologias alternativas para a comparação dos grafos de dependência merece ser investigada, para que os níveis de similaridade medidos permitam a eliminação da necessidade do estabelecimento de padrões mínimos de similaridade medidos a partir da comparação do grafo de referência original e o grafo e referência gerado pela redução aprimorada. Alternativamente, o algoritmo de comparação atual deve ser modificado para que seu desempenho na comparação entre grafos com diferentes quantidades de vértices seja o mesmo apresentado pela comparação entre grafos com quantidades de vértices iguais.

Por último, para o aprimoramento do processo de comparação entre os diferentes grafos de referência, poderia se beneficiar da introdução de um procedimento de filtragem onde apenas os grafos de referência com características mais compatíveis com o grafo de dependência do programa investigado seriam selecionados para o processo de comparação. O uso de abordagens baseadas em aprendizado de máquina ou outros métodos de mineração de dados podem ser empregados para esta tarefa.

Referências

[1] W. Stallings and D. Vieira, *Criptografia e segurança de redes: princípios e pràticas*. Pearson Prentice Hall, 2008.

[2] E. Skoudis, *Malware: Fighting malicious code*. Prentice Hall Professional, 2004.

[3] H. Yin, D. Song, and M. Egele, “Panorama: capturing system-wide information flow for malware detection and analysis,” *Proc. 14th …*, pp. 116–127, 2007.

[4] P. Wurzinger, L. Bilge, T. Holz, J. Goebel, C. Kruegel, and E. Kirda, “Automatically generating models for botnet detection,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 5789 LNCS, pp. 232–249, 2009.

[5] K. Griffin, S. Schneider, X. Hu, and T. C. Chiueh, “Automatic generation of string signatures for malware detection,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 5758 LNCS, pp. 101–120, 2009.

[6] M. Colajanni, D. Gozzi, and M. Marchetti, “Collaborative architecture for malware detection and analysis,” *IFIP Int. Fed. Inf. Process.*, vol. 278, pp. 79–93, 2008.

[7] M. Colajanni, M. Marchetti, and M. Messori, “Selective and Early Threat Detection in Large Networked Systems,” *2010 10th IEEE Int. Conf. Comput. Inf. Technol.*, no. Cit, pp. 604–611, Jun. 2010.

[8] K. Rieck, T. Holz, and C. Willems, “Learning and classification of malware behavior,” *Detect. Intrusions …*, 2008.

[9] D. Bruschi, L. Martignoni, and M. Monga, “Code Normalization for Self-Mutating Malware,” *IEEE Secur. Priv. Mag.*, vol. 5, no. 2, pp. 46–54, Mar. 2007.

[10] J.-M. Borello and L. Mé, “Code obfuscation techniques for metamorphic viruses,” *J. Comput. Virol.*, vol. 4, no. 3, pp. 211–220, Feb. 2008.

[11] X. Hu, T. Chiueh, and K. G. Shin, “Large-scale malware indexing using function-call graphs,” *Proc. 16th ACM Conf. Comput. Commun. Secur. - CCS ’09*, p. 611, 2009.

[12] W. Wong and M. Stamp, “Hunting for metamorphic engines,” *J. Comput. Virol.*, vol. 2, no. 3, pp. 211–229, Nov. 2006.

[13] G. Jacob, H. Debar, and E. Filiol, “Malware detection using attribute-automata to parse abstract behavioral descriptions,” *arXiv Prepr. arXiv0902.0322*, no. 1, Feb. 2009.

[14] M. F. Cozzolino, G. B. Martins, E. Souto, and F. E. G. Deus, “Detecção de variações de malware metamórfico por meio de normalização de código e identificação de subfluxos,” in *Anais do XII Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais*, 2012, pp. 30–43.

[15] K. Kim and B.-R. Moon, “Malware detection based on dependency graph using hybrid genetic algorithm,” *Proc. 12th Annu. Conf. Genet. Evol. Comput. - GECCO ’10*, p. 1211, 2010.

[16] W. Baker, A. Hutton, C. D. Hylender, C. Novak, P. Tippett, D. Ph, C. Chang, E. Gentry, K. Long, D. Todd, J. A. Valentine, and C. Dismukes, “2011 Data Breach Investigations Report 2011 Data Breach Investigations Report (DBIR),” 2011.

[17] C. Liu, C. Chen, J. Han, and P. Yu, “GPLAG: detection of software plagiarism by program dependence graph analysis,” *Proc. 12th ACM SIGKDD …*, pp. 872–881, 2006.

[18] T. Yagi, N. Tanimoto, T. Hariu, and M. Itoh, “Investigation and analysis of malware on websites,” in *Web Systems Evolution (WSE), 2010 12th IEEE International Symposium on*, 2010, pp. 73–81.

[19] H. C. Jeong, “Malware Web Site Detection.” [Online]. Available: http://www.cert.org/archive/pdf/MCFinder_KrCERTCC.pdf.

[20] F. Kai, S. Jianhua, and C. Hao, “WSProxy: Detecting and Fighting Malicious Websites,” *2011 Int. Conf. Bus. Comput. Glob. Informatiz.*, pp. 649–652, Jul. 2011.

[21] Barossa Community Co-operative Store, “Pandalabs Annual Report 2014,” pp. 1–28, 2014.

[22] G. Tahan, C. Glezer, Y. Elovici, and L. Rokach, “Auto-Sign: An automatic signature generator for high-speed malware filtering devices,” *J. Comput. Virol.*, vol. 6, no. 2, pp. 91–103, 2010.

[23] M. Morin, “The Financial Impact of Attack Traffic on Broadband Networks,” *IEC Annu. Rev. Broadband Commun.*, pp. 11–14, 2006.

[24] P. Security, “Annual Report PandaLabs 2009,” 2009. [Online]. Available: http://www.pandasecurity.com/img/enc/Annual_Report_PandaLabs_2009.pdf.

[25] B. Watkins, “The Impact of Cyber Attacks on the Private Sector,” no. August, pp. 1–11, 2014.

[26] “Symantec: Internet Security Threat Report, Volume 16,” *Symantec*. [Online]. Available: http://www.symantec.com/business/threatreport/build.jsp.

[27] M. F. Botacin, A. Grégio, and P. L. De Geus, “Uma Visão Geral do Malware Ativo no Espaço Nacional da Internet entre 2012 e 2015,” pp. 1–10, 2015.

[28] A. E. Brill and S. Petreska, “Are Cyber Criminals Competing at the Olympics?,” *Free. From Fear, Ed.*, vol. 10, 2015.

[29] J. Lafloufa, “Hackativismo: crime cibern{é}tico ou leg{’\i}tima manifesta{ç}{ã}o digital?,” *ComCi{ê}ncia*, no. 131, p. 0, 2011.

[30] K. Ask, “Automatic Malware Signature Generation,” 2006.

[31] J. Newsome, B. Karp, and D. Song, “Polygraph: Automatically Generating Signatures for Polymorphic Worms,” *2005 IEEE Symp. Secur. Priv.*, pp. 226–241, 2005.

[32] G. Notoatmodjo, “Detection of Self-Mutating Computer Viruses,” *virii.es*, pp. 1–17.

[33] S. W. Hsiao, Y. S. Sun, M. C. Chen, and H. Zhang, “Behavior profiling for robust anomaly detection,” *Proc. - 2010 IEEE Int. Conf. Wirel. Commun. Netw. Inf. Secur. WCNIS 2010*, pp. 465–471, 2010.

[34] J. Newsome and D. Song, “Dynamic Taint Analysis for Automatic Detection , Analysis , and Signature Generation of Exploits on Commodity Software and Signature Generation of Exploits on Commodity Software,” 2005.

[35] B. B. Rad, M. Masrom, and S. Ibrahim, “Camouflage in Malware : from Encryption to Metamorphism,” vol. 12, no. 8, pp. 74–83, 2012.

[36] P. O’Kane, S. Sezer, and K. McLaughlin, “Obfuscation: The Hidden Malware,” *IEEE Secur. Priv.*, vol. 9, no. 5, pp. 41–47, 2011.

[37] I. You and K. Yim, “Malware Obfuscation Techniques : A Brief Survey,” *2010 Int. Conf. Broadband, Wirel. Comput. Commun. Appl.*, pp. 297–300, 2010.

[38] B. Rad and M. Masrom, “Metamorphic virus variants classification using opcode frequency histogram,” *arXiv Prepr. arXiv1104.3228*, vol. I, no. Volume I, pp. 147–155, 2011.

[39] A. Karnik, S. Goswami, and R. Guha, “Detecting Obfuscated Viruses Using Cosine Similarity Analysis,” *First Asia Int. Conf. Model. Simul.*, 2007.

[40] M. Dacier, F. Pouget, and H. Debar, “Honeypots: Practical means to validate malicious fault assumptions,” *Proc. - IEEE Pacific Rim Int. Symp. Dependable Comput.*, pp. 383–388, 2004.

[41] M. J. Ranum, “A Whirlwind Introduction to Honeypots What is a honeypot ? Cool Record : Two Kinds of Honeypots,” pp. 1–18, 2002.

[42] I. D. Systems, “Honeynet Overview Honeynet Architecture,” pp. 1–4, 2010.

[43] E. Piva and P. de Geus, “Using Virtal Machines to increase honeypot security,” *V Simpósio Bras. em Segurança da …*, pp. 249–252, 2005.

[44] M. Dalla Preda, R. Giacobazzi, and S. Debray, “Unveiling metamorphism by abstract interpretation of code properties,” *Theor. Comput. Sci.*, vol. 577, pp. 74–97, 2015.

[45] C. C. Zou and R. Cunningham, “Honeypot-aware advanced botnet construction and maintenance,” *Proc. Int. Conf. Dependable Syst. Networks*, vol. 2006, pp. 199–208, 2006.

[46] A. A. E. Elhadi, M. A. Maarof, B. I. A. Barry, and H. Hamza, “Enhancing the detection of metamorphic malware using call graphs,” *Comput. Secur.*, vol. 46, pp. 62–78, 2014.

[47] J. Ferrante, K. J. Ottenstein, and J. D. Warren, “The program dependence graph and its use in optimization,” *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, Jul. 1987.

[48] J. Krinke, “Identifying similar code with program dependence graphs,” *Proc. Eighth Work. Conf. Reverse Eng.*, pp. 301–309, 2001.

[49] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979.

[50] OllyDbg, “OllyDbg v1.10,” 2014. [Online]. Available: http://www.ollydbg.de.

[51] Hex-Rays, “IDA: About,” 2014. [Online]. Available: http://www.hex-rays.com/products/ida/index.shtml.

[52] I. M. Bomze, M. Budinich, P. M. Pardalos, and M. Pelillo, “The maximum clique problem,” in *Handbook of combinatorial optimization*, Springer US, 1999, pp. 1–74.

[53] R. W. Floyd, “Algorithm 97: shortest path,” *Commun. ACM*, vol. 5, no. 6, p. 345, 1962.

[54] S. Warshall, “A theorem on boolean matrices,” *J. ACM*, vol. 9, no. 1, pp. 11–12, 1962.

[55] J. Konc and D. Janezic, “An improved branch and bound algorithm for the maximum clique problem,” *MATCH Commun. Math. Comput. Chem.*, vol. 58, pp. 569–590, 2007.

[56] A. N. Crespo, O. J. Da Silva, C. A. Borges, C. F. Salviano, M. De Teive, A. Junior, and M. Jino, “Uma Metodologia para Teste de Software no Contexto da Melhoria de Processo,” no. Simpósio Brasileiro de Qualidade de Software, pp. 271–285, 2004.

[57] P. Foggia, M. Vento, and I. Elettrica, “Challenging Complexity of Maximum Common Subgraph Detection Algorithms : A Performance Analysis of Three Algorithms on a Wide Database of Graphs Donatello Conte,” vol. 11, no. 1, pp. 99–143, 2007.

[58] G. Martins, R. De Freitas, and E. Souto, “Virtual Structures and Heterogeneous Nodes in Dependency Graphs for Detecting Metamorphic Malware,” in *33rd IEEE International Performance, Computing, and Communications Conference (IPCCC 2014)*, 2014.