Gilbert B. Martins[1], Paulo dos Santos[2], Vitor Danrley[2], Eduardo Souto[2], Rosiane de Freitas[2]

[1]Campus Distrito Industrial – Instituto Federal do Amazonas (IFAM)
Manaus – AM – Brasil

[2]Instituto de Computação – Universidade Federal do Amazonas (UFAM)
Manaus – AM – Brasil

{gilbert.breves, paulohsantosol}@gmail.com,
{vdms, esouto, rosiane}@icomp.ufam.edu.br

In order to correctly identify metamorphic malicious code, different approaches try to model structural characteristics that remain valid even after the application code obfuscation techniques. One approach is based on comparing dependency graphs generated from suspicious code with a baseline graphs generated from previously identified malicious code. However, as the graph comparison process is a NP-Hard problem, the development of methodologies comparison that makes possible this process of identification is required. This article presents the results from a methodology that uses the concepts of differentiation of vertices and adapted topological sort, to propose a metric measuring maximum subgraph isomorphism.

Resumo. Com o objetivo de identificar corretamente códigos maliciosos metamórficos, diferentes abordagens procuram modelar características estruturais que se mantêm válidas, mesmo após a aplicação de técnicas de ofuscação de código. Uma destas abordagens se baseia na comparação de grafos de dependência, extraídos de códigos suspeitos com uma base de grafos referência gerada a partir de códigos maliciosos previamente identificados. Entretanto, como o processo de comparação de grafos é um problema NP-Difícil, se faz necessário o desenvolvimento de metodologias de comparação que tornem viável este processo de identificação. Este artigo apresenta os resultados obtidos a partir de uma metodologia que usa os conceitos de diferenciação de vértices e ordenação topológica adaptada, para propor uma métrica de medição de máximo subgrafo isomorfismo.

1. Introdução

O aumento das atividades maliciosas e a inclusão de recursos de ofuscação de código que mascaram a identidade dos programas maliciosos (*malware*) tornam contínua a necessidade de desenvolvimento de técnicas alternativas de identificação, capazes de estabelecer corretamente a identidade de um código suspeito [Barossa Community Co-operative Store 2014; Borello and Mé 2008].

A princípio, a simples extração de trechos de código de *malware* previamente identificados, também conhecidos como *assinaturas*[Jacob et al. 2009] [Newsome et al. 2005], era suficiente para identificar corretamente outras cópias de um mesmo programa malicioso. Qualquer código suspeito era comparado com uma base de assinaturas de *malware* e, caso uma dessas assinaturas fosse encontrada em uma parte do código sob análise, a identidade do programa era estabelecida e a ameaça tratada.

Entretanto, como este processo de identificação exige um casamento perfeito entre uma parte do código suspeito e uma das assinaturas, os desenvolvedores de códigos maliciosos começaram a procurar formas de alterar a codificação original de suas criações, de maneira a que a identificação por assinaturas pudesse ser inviabilizada.

A princípio, as técnicas empregadas para este fim eram baseadas no uso de criptografia do código original, que era posteriormente reconstruído apenas no momento do ataque ao sistema alvo. A estrutura básica deste tipo de código furtivo envolve dois componentes distintos: um motor criptográfico e um corpo criptografado do *malware* propriamente dito [Rad et al. 2012] [O'Kane et al. 2011][You and Yim 2010].

O motor criptográfico tem duas responsabilidades principais: *i)* a restauração do corpo criptografado ao seu estado original, para que o *malware* possa agir de acordo com os objetivos para o qual foi construído; e *ii)* a criação de um novo corpo criptografado, baseado numa chave aleatória, no momento em que ocorre o processo de propagação. É este processo de criptografia com chave aleatória que possibilita criação de uma nova instância deste código malicioso e que a torna incompatível com qualquer assinatura gerada a partir do corpo criptografado de outra instância em particular.

Os primeiros *malware* que utilizaram essa técnica empregavam um único motor criptográfico, isso permitiu que as assinaturas geradas a partir deste componente fossem suficientes para identificar estes códigos maliciosos. Para fugir desta limitação, sucessivas alterações modificações na técnica original foram produzidas, com o passar do tempo. Iniciando com o uso de um conjunto de motores criptográficos distintos, escolhidos aleatoriamente no momento que o processo de propagação (Oligomorfismo), e culminando a geração de múltiplos motores criptográficos, com o emprego de técnicas de ofuscação de código para este componente do código malicioso (Polimorfismo) [Rad et al. 2012] [O'Kane et al. 2011][You and Yim 2010].

Apesar da técnica de polimorfismo ter o potencial para criar uma quantidade infinita de instâncias de *malware*, ela ainda compartilha o mesmo problema de 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 (*sandbox*), onde o código reconstruído é carregado e analisado, sem que qualquer tipo de contaminação venha a ocorrer [You and Yim 2010].

Percebendo que este conjunto de técnicas de furtividade tinha atingido o 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.

O efeito de metamorfismo é o obtido pela ação em conjunto de técnicas muito simples. Operações como: *i)* a inserção de instruções e variáveis irrelevantes, também conhecida como inserção de código lixo; *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 [Bruschi et al. 2007], são capazes modificar o código o suficiente para que sua assinatura original seja invalidada.

Diferentes abordagens foram propostas para lidar com este problema [Cozzolino et al. 2012; Griffin et al. 2009; Hu et al. 2009; Jacob et al. 2008, 2009; Kim and Moon 2010]. Algumas dessas abordagens se baseiam em modelagens que procuram invalidar as modificações introduzidas pelo metamorfismo [ ] [ ] [ ] [ ], enquanto outras se baseiam na construção de grafos de dependência[You and Yim 2010] para o mapeamento das relações de interdependência entre componentes de código. Exemplos de elementos que podem ser mapeados por estes grafos incluem: chamadas de funções [Hu et al. 2009], uso de variáveis por instruções [Kim and Moon 2010] e chamadas bibliotecas de APIs (*Aplication Program Interfaces*) [Elhadi et al. 2014].

Como todas as abordagens baseadas em grafos de dependência lidam com o problema NP-Difícil da identificação do nível de similaridade entre grafos[Foggia et al. 2007][Eppstein 1999][Johnson 2005], é fundamental propor uma metodologia de comparação eficiente e uma métrica de medição do nível de similaridade presente nos grafos empregados no processo de identificação.

Este trabalho propõe uma metodologia para identificação de códigos metamórficos baseada na geração de grafos de dependência de referência, gerados a partir da identificação das partes mais relevantes de grafos de dependência extraídos com base na análise estática de códigos executáveis. O processo de identificação emprega uma metodologia de medição que mensura o máximo isomorfismo de subgrafo presente entre as estruturas comparadas, em conjunto com um algoritmo genético modificado que leva em consideração características topológicas dos grafos gerados a partir dos programas executáveis. Avaliações realizadas, tanto com coleções grafos sintéticos como com grafos gerados partir de versões metamórficas de *malware* reais, demonstram melhorias no processo de identificação de códigos maliciosos metamórficos.

O restante deste artigo está organizado da seguinte forma. A seção 2 fornece alguns trabalhos relacionados ao problema de detecção de *malware* metamórfico baseados em grafos de dependência. A seção 3 apresenta uma metodologia para identificação de *malware* metamórfico baseada em grafos de dependência. A seção 4 detalha o processo de diferenciação de tipos de vértices que dá suporte a todos os

aprimoramentos propostos neste artigo. A seção 5 detalha o processo de comparação de grafos de dependência proposto neste artigo. A seção 6 fornece resultados experimentais. A seção 7 apresenta as conclusões e trabalhos futuros.

2. Conceitos Básicos e Trabalhos Relacionados

Grafos de dependência podem ser construídos de forma a modelar as inter-relações entre diversos componentes. Códigos executáveis podem ser analisados de forma a modelar as relações de dependência existentes entre todas as chamadas da função [Hu et al. 2009]. Cada vértice v_i está associado a uma função e uma aresta $v_a v_b$ é criada sempre que no corpo da função v_a existir uma chamada para a função v_b. A partir deste ponto, o processo de identificação compara o grafo gerado com uma base de referência, permitindo a identificação do *malware*. A maior limitação deste processo está associada à dificuldade de mapear corretamente as funções criadas diretamente no código do programa.

Para detecção de códigos maliciosos presentes em *scripts* [Kim and Moon 2010], o código suspeito é analisado para a extração de um grafo de dependência baseado nas variáveis que cada instrução manipula. O grafo gerado passa ainda por um processo de redução que elimina elementos gerados a partir da ação do metamorfismo do código. O processo de detecção procura de encontrar o máximo isomorfismo de subgrafo entre o grafo reduzido e um conjunto de grafos de referência extraídos de malware previamente identificados. Entretanto, a definição de níveis de similaridade usados para definir uma identificação positiva não é muito clara.

Outra proposta [Elhadi et al. 2014], trata o problema de identificação por meio do uso de grafos que modelam chamadas bibliotecas de APIs. Este modelo se concentra 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. 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 passam a ser comparados através da medição do nível de similaridade entre seus grafos de chamada. 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.

De forma geral, todas essas abordagens não fazem qualquer tipo de distinção entre a natureza de cada vértice, durante a execução deste processo de comparação de grafos. Por exemplo, o algoritmo de [Kim and Moon 2010] propõe o uso de um algoritmo genético que consome parte de seu tempo comparando vértices de processamento com vértices de decisão, desperdiçando tempo de processamento desnecessariamente. A proposta apresentada neste artigo pretende eliminar este desperdício, pelo uso ativo da classificação de vértices para que elementos de natureza distinta não sejam comparados durante o processo de identificação, sem que para isto seja necessário alterar o modelo de codificação utilizado para armazenar os grafos na base de referência.

3. Metodologia de Identificação de Códigos Metamórficos Executáveis

Em uma parte anterior da pesquisa que originou este trabalho [Martins et al. 2014] propôs um processo de identificação de códigos executáveis metamórficos, baseada na utilização de grafos de dependência. Este processo pode ser descrito resumidamente como sendo composto por quatro etapas principais:

i. Reconstrução de código *assembly*, onde o programa executável passa por um processo de engenharia reversa para a obtenção do seu código equivalente em linguagem *assembly*. Esta etapa pode ser executada com o auxílio de programas como OllyDbg [1] e IDA Pro [2].

ii. Geração do grafo de dependência, onde o programa gerado na etapa anterior é analisado e então usado como base para a geração do grafo de dependência.

iii. Redução do grafo, usado para reduzir o grafo de dependência obtido na etapa anterior. Partes do código onde o controle de fluxo nunca irá passar são removidas. De acordo com nossa proposta, um tratamento adicional de redução deve ser executado para os grafos que constituírem a base de referência.

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

É importante destacar que o processo de geração dos grafos que constituem a base de referência também segue as três primeiras etapas do processo de identificação, com a diferença de que a etapa de redução do grafo passa por um processo aprimorado de redução, desenvolvido para identificar os elementos estruturais mais relevantes do grafo, com base na diferenciação dos tipos de vértices presentes e suas inter-relações.

Este trabalho apresenta um aprimoramento dessa abordagem, introduzindo o conceito de diferenciação de vértices e de ordenação topológica, no processo de comparação de grafos. Além disso, serão apresentados resultados comparativos com ferramentas comerciais de detecção de *malware*. Para um melhor entendimento destas contribuições, o conceito de diferenciação de vértices é explicado com detalhes na Seção 4, enquanto a Seção 5 demonstra como os conceitos de ordenação topológica foram introduzidos no processo de comparação entre grafos de dependência.

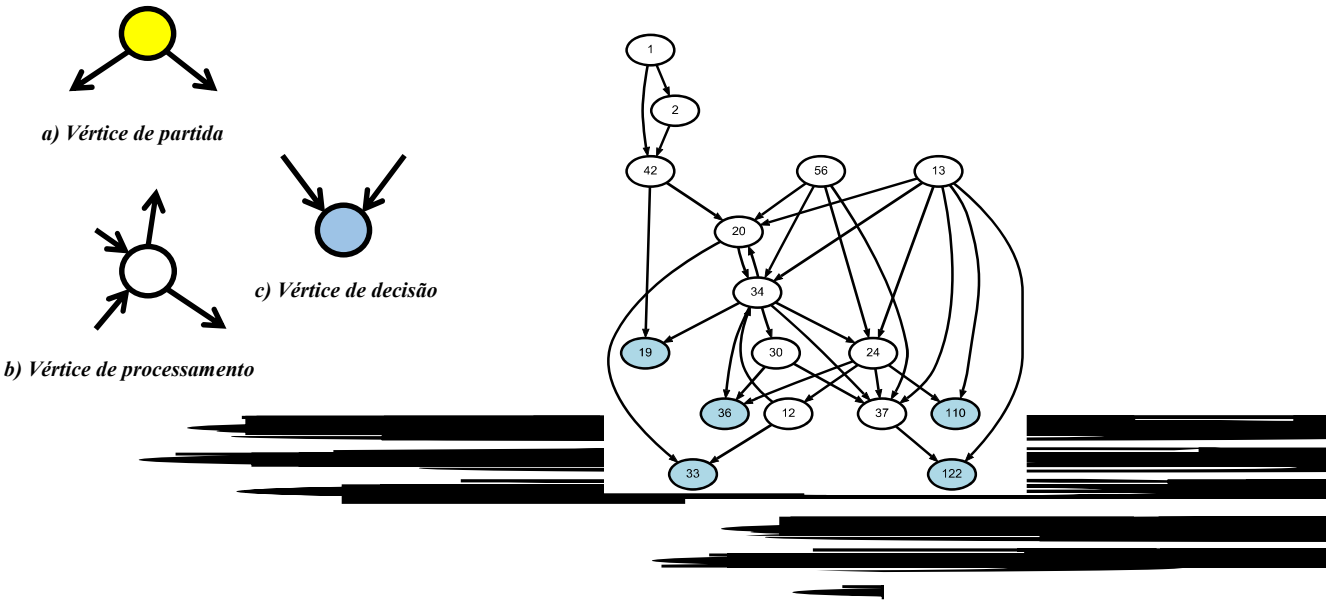
4. Diferenciação dos Vértices em Grafos de Dependência

Este trabalho propõe uma diferenciação dos vértices de um grafo de dependência com base nas análises estruturais dos vértices pertencentes a grafos de dependência extraídos de programas executáveis e da correlação destes vértices com as instruções presentes nos códigos *assembly,* usados para gerar estes grafos.

[1] http://www.ollydbg.de.

[2] http://www.hex-rays.com/products/ida/index.shtml.

Portanto, a partir dessa análise, estes vértices podem ser classificados em três categorias distintas, conforme: a) vértices de partida; b) vértices de processamento; e c) vértices de decisão. Estes vértices são ilustrados na Figura 1.



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 onde o programa original poderia iniciar o seu processamento. Na Figura 2, os nós 1, 13 e 56 são exemplos deste tipo de vértice.

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 neste trabalho, fazem parte desta categoria. Na Figura 2, 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. Exemplos deste tipo de vértice incluem os nós 19, 33, 36, 110 e 122 na Figura 2.

Como programas escritos em *assembly* não possuem instruções de controle de alto nível como `if-then-else` e `do-while`, é através de instruções `CMP` 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.

O processo usado para identificar os vértices de decisão mais relevantes para o programa modelado pode ser resumido 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 [Bomze et al. 1999] 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. Detalhes de cada uma dessas etapas podem ser encontrados em um trabalho anterior dos autores [Martins et al. 2014].

Assim, o grafo gerado por este processo de serve como referência para identificação de versões metamórficas do mesmo código que o originou, através de um processo de comparação entre grafos.

5. Comparação dos Grafos de Dependência

Os algoritmos de comparação de grafos visam 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.

A diferenciação entre os tipos de vértices, presentes nos grafos de dependência, também é empregada para aprimorar o processo de comparação e identificação 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. Esta diferenciação contribui para a precisão dos resultados porque o processo de comparação usa esta informação para evitar que vértices de natureza distinta sejam comparados entre si. O conjunto total de propriedades que suportam esta metodologia é apresentado na Tabela 1.

Descrição	**Equação**
Definições iniciais	$G_1=(V_1, E_1)$, $G_2=(V_2, E_2)$ e $\|V_1\| < \|V_2\|$
Detalhamento dos vértices	$V=V_d, \cup V_p$
Subgrafo G'	$G'=(V', E') \mid G' \subset G, V' \subset V, E' \subset E$
Subgrafos comparados	$G_1' \subset G_1$, $G_2' \subset G_2 \mid \|V_{d1}'\| = \|V_{d2}'\|, \|V_{p1}'\| = \|V_{p2}'\|$
Função de busca de uma aresta e em um conjunto de arestas E	$I(e,E) = \begin{cases} 1, & se \quad e \in E \\ 0, & caso \quad contrário \end{cases}$
Cálculo da similaridade entre G_1 e G_2	$similaridade(G_1', G_2') = \frac{\sum_{e \in E_1'} I(e, E_2') + \sum_{e \in E_2'} I(e, E_1')}{\|E_1'\| + \|E_2'\|}$

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 igual, conforme as propriedades apresentadas na Tabela 1. 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.

A função de cálculo da similaridade entre os grafos gera uma pontuação de similaridade. Esta pontuação é usada para determinar se existe ou não contaminação por *malware* no programa analisado.

O processo base de comparação das amostras utiliza um algoritmo genético, que deixa o grafo G_1 inalterado e gera diferentes arranjos de vértices do grafo $G2$ que, a partir daí, são comparadas com o grafo G_1. Cada um desses arranjos representa um membro de uma população de 100 elementos. Esta população inicial é renovada à 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.

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

Uma das metas deste trabalho é introduzir o processo de diferenciação de vértices no momento da comparação entre grafos, 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 um processo de *comparação segmentada com restrição de escopo*.

5.2 Segmentação por Tipo de Vértice.

Antes de iniciar o processo de comparação, 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 receberão 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. A Figura 3 ilustra a diferença entre a disposição de cromossomos sem o uso da segmentação (Figura 3.a) e com o uso da segmentação (Figura 3.b), para o grafo representado na Figura 2.

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

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

b) 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

5.3 Comparação Segmentada com Restrição de Escopo.

Na comparação segmentada com restrição de escopo (ou comparação por escopo), empregamos as mesmas estratégias genéticas utilizadas na abordagem de Kim e Moon [Kim and Moon 2010]. 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.

Entretanto, os vértices de processamento comparados são 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) Nesta etapa, os primeiros vértices de processamento inseridos têm 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 4 ilustra a diferença de disposição entre o processo de segmentação (Figura 4.a) e o processo de restrição de escopo (Figura 4.b), para o grafo da Figura 2.

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

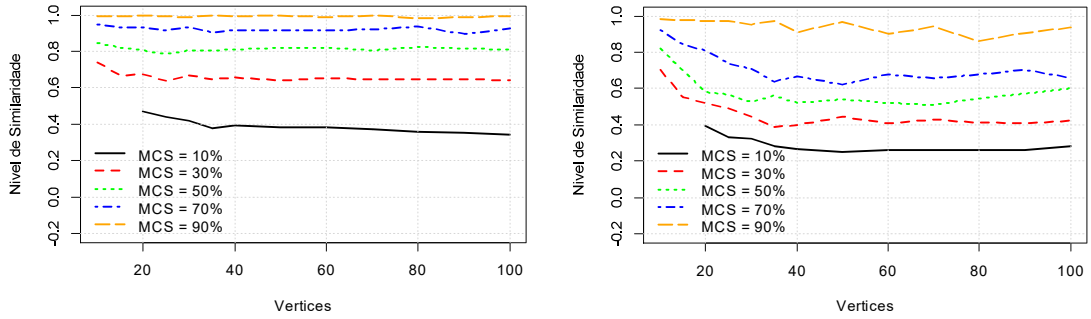
19	33	36	110	122	34	42	12	20	24	30	13	37	1	2	56

6. Resultados

Inicialmente, para avaliar o desempenho da técnica de comparação por tipo e comparação por restrição de escopo, foi empregada uma base de dados sintética com o os valores de MCS (*Maximum Common Subgraph* ou maior subgrafo comum) conhecidos. Esta base é formada por um conjunto de pares de grafos com uma quantidade mínima de vértices e arestas iguais previamente estabelecidas[Foggia et al. 2007]. O objetivo é avaliar o impacto que a introdução comparação com restrição de escopo é capaz de produzir na execução do algoritmo genético de comparação.

A partir dos resultados obtidos, é possível perceber que os resultados da comparação sem o uso da restrição de escopo (Figura 5.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 a 70% atingiram pontuações mais condizentes com as características apresentadas pelos pares de grafos analisados.



Assim, apesar das faixas de MCS estarem todas separadas por um fator linear (cada faixa foi gerada em saltos de 20%), as faixas de pontuação obtidas tiveram um comportamento não-linear, quando relacionadas umas às outras

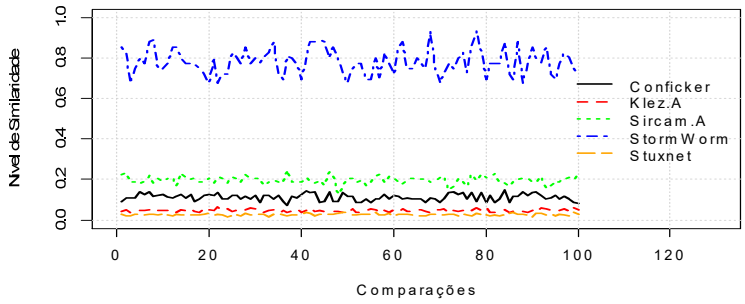
Nos resultados obtidos com a utilização da comparação com restrição de escopo (Figura 5.b), os níveis de pontuação obtidos pelas amostras com menor quantidade de vértices, foram similares àqueles obtidos na primeira avaliação. Entretanto, à medida que a quantidade de vértices aumentou, o nível de pontuação 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.

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 projeto de pesquisa [Martins et al. 2014]. 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, garantindo 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 6.

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, 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. 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 2.

Além destes testes, também foram avaliadas as ocorrências de identificações de falsos positivos. Os resultados 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 3.

Testes de falso positivo foram realizados 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 taxa de falsos positivos gerada foi de 7%. Infelizmente, a eficiência do algoritmo de comparação foi afetada pela variação entre a quantidade de vértices das amostras sem contaminação e a quantidade de vértices do grafo do *malware*.

Finalmente, algumas das amostras metamórficas foram submetidas à ferramenta Vírus Total[3], que avalia cada arquivo a mais de cinquenta ferramentas de antivírus comerciais. Apesar das ferramentas comerciais implementarem diferentes estratégias para tratar o problema do *malware* metamórfico, elas ainda estão longe de serem soluções efetivas. 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

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

dessas ferramentas seja completa. A Tabela 4 apresenta alguns dos resultados obtidos onde é indicada a quantidade de testes e de identificações positivas obtidas.

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%

Malware Base	Amostras	Referência	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%

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

7. Conclusões

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 introdução do processo de diferenciação de vértices no processo de redução permitiu estabelecer uma base clara para os níveis de pontuação necessários para estabelecer uma identificação positiva. Além disso, quando a diferenciação de vértices foi combinada com características de ordenação topológica, o processo de comparação também foi aprimorado, com a geração de resultados mais fiéis àqueles observados nos testes com uma base de grafos sintéticos. Finalmente, quando comparados com as ferramentas comerciais o processo de identificação obteve resultados bem superiores, atestando o potencial da metodologia proposta.

Como trabalhos futuros, o emprego de metodologias alternativas para a comparação dos grafos de dependência merece ser investigada. Além disso, o algoritmo de comparação atual pode ser modificado para que seu desempenho na comparação entre grafos com diferentes quantidades de vértices seja mais próximo àquele apresentado pela comparação entre grafos com quantidades de vértices iguais.

Agradecimentos

Este trabalho foi parcialmente financiado pela Fundação de Amparo à Pesquisa do Estado do Amazonas – FAPEAM (processo 062.03178/2012), e pela CAPES.

Referências

Barossa Community Co-operative Store (2014). Pandalabs Annual Report 2014. p. 1–28.

Bomze, I. M., Budinich, M., Pardalos, P. M. and Pelillo, M. (1999). The maximum clique problem. *Handbook of combinatorial optimization*. Springer US. p. 1–74.

Borello, J.-M. and Mé, L. (21 feb 2008). Code obfuscation techniques for metamorphic viruses. *Journal in Computer Virology*, v. 4, n. 3, p. 211–220.

Bruschi, D., Martignoni, L. and Monga, M. (mar 2007). Code Normalization for Self-Mutating Malware. *IEEE Security and Privacy Magazine*, v. 5, n. 2, p. 46–54.

Cozzolino, M. F., Martins, G. B., Souto, E. and Deus, F. E. G. (2012). 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*.

Elhadi, A. A. E., Maarof, M. A., Barry, B. I. A. and Hamza, H. (2014). Enhancing the detection of metamorphic malware using call graphs. *Computers & Security*, v. 46, p. 62–78.

Eppstein, D. (9 nov 1999). Subgraph Isomorphism in Planar Graphs and Related Problems. v. 3, n. 3, p. 27.

Foggia, P., Vento, M. and Elettrica, I. (2007). Challenging Complexity of Maximum Common Subgraph Detection Algorithms : A Performance Analysis of Three Algorithms on a Wide Database of Graphs Donatello Conte. v. 11, n. 1, p. 99–143.

Griffin, K., Schneider, S., Hu, X. and Chiueh, T. C. (2009). Automatic generation of string signatures for malware detection. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, v. 5758 LNCS, p. 101–120.

Hu, X., Chiueh, T. and Shin, K. G. (2009). Large-scale malware indexing using function-call graphs. *Proceedings of the 16th ACM conference on Computer and communications security - CCS '09*, p. 611.

Jacob, G., Debar, H. and Filiol, E. (21 feb 2008). Behavioral detection of malware: from a survey towards an established taxonomy. *Journal in Computer Virology*, v. 4, n. 3, p. 251–266.

Jacob, G., Debar, H. and Filiol, E. (2 feb 2009). Malware detection using attribute-automata to parse abstract behavioral descriptions. *arXiv preprint arXiv:0902.0322*, n. 1.

Johnson, D. S. (1 jul 2005). The NP-completeness column. *ACM Transactions on Algorithms*, v. 1, n. 1, p. 160–176.

Kim, K. and Moon, B.-R. (2010). Malware detection based on dependency graph using hybrid genetic algorithm. *Proceedings of the 12th annual conference on Genetic and evolutionary computation - GECCO '10*, p. 1211.

Martins, G., Souto, E., Freitas, R. De and Feitosa, E. (2014). Estruturas Virtuais e Diferenciação de Vértices em Grafos de Dependência para Detecção de Malware Metamórfico. *lbd.dcc.ufmg.br*, p. 237–250.

Newsome, J., Karp, B. and Song, D. (2005). Polygraph: Automatically Generating Signatures for Polymorphic Worms. *2005 IEEE Symposium on Security and Privacy (S&P'05)*, p. 226–241.

O'Kane, P., Sezer, S. and McLaughlin, K. (2011). Obfuscation: The Hidden Malware. *IEEE Security & Privacy*, v. 9, n. 5, p. 41–47.

Rad, B. B., Masrom, M. and Ibrahim, S. (2012). Camouflage in Malware : from Encryption to Metamorphism. v. 12, n. 8, p. 74–83.

You, I. and Yim, K. (2010). Malware Obfuscation Techniques : A Brief Survey. *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*, p. 297–300.

