Estruturas Virtuais e Diferenciação de Vértices em Grafos de Dependência para Detecção de *Malware* Metamórfico.

Gilbert B. Martins, Eduardo Souto, Rosiane de Freitas, Eduardo Feitosa

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

{gilbert.martins, esouto, rosiane, efeitosa}@icomp.ufam.edu.br

Abstract. *This paper presents a methodology for identifying metamorphic malware based on the comparison of dependency graphs stored in a reference. On the strength of the structural differentiation of the vertices and the addition of virtual structures, the proposed methodology is able to identify and eliminate non-relevant elements of the original reference graph, reducing the size of the reference database and improving the results obtained during the comparison of the graphs. To validate this, is presented the comparison of results generated by the proposed approach with those from a reference method in the identification of W32.Evol and W32.Polip metamorphic malwares.*

Resumo. Este artigo apresenta uma metodologia de identificação de *malware* metamórfico baseada na comparação de grafos de dependência armazenados numa base de referência. Em função da diferenciação estrutural dos vértices e da adição de estruturas virtuais, a metodologia proposta é capaz de identificar e eliminar os elementos não relevantes do grafo de referência original, reduzindo o tamanho da base de referência e melhorando a variância nos resultados obtidos durante a comparação entre os grafos. Para validar isto, é apresentada a comparação dos resultados obtidos com aqueles gerados por uma metodologia de referência, na identificação dos *malware* metamórficos W32.Evol e W32.Polip.

1. Introdução

A última década tem enfrentado um aumento significativo no número e na sofisticação dos ataques digitais baseados em *malwares*[1] [Baker et al. 2011]. Fatores como o uso da Internet para a distribuição massificada e a capacidade de auto-propagação por redes locais, aumentam consideravelmente o grau de dificuldade do combate a estas ameaças. Para tratar este problema, é comum fazer uso de uma base de dados contendo trechos de código extraídos de cada *malware*, que associada a um processo de varredura de programas suspeitos, permite a identificação de códigos maliciosos [Karin 2006]. É importante salientar que, para o processo de identificação ter sucesso, o trecho de código

[1] O termo *malware* (do inglês, *malicious software*) é usado para classificar um software destinado a se infiltrar em um sistema de computador alheio de forma ilícita, com o intuito de causar algum dano ou roubo de informações (confidenciais ou não).

selecionado como assinatura deve ser único o suficiente para ser encontrado apenas no *malware* que se pretende identificar [Moura e Rebiha 2009].

Apesar de eficiente, a identificação pelo uso de assinaturas só classificará como *malware*, programas que possuírem, dentro de sua codificação, um trecho de código que seja idêntico a uma das assinaturas catalogadas. Tal premissa tem sido explorada pelos desenvolvedores de códigos maliciosos, que empregam técnicas de ofuscação de código para dificultar o processo de detecção [Borello e Mé 2008][Notoatmodjo 2010]. Tais técnicas têm como objetivo mascarar a identidade do programa malicioso e se baseiam na modificação da sequência original de instruções sem prejuízo à funcionalidade original dos trechos alterados. Exemplos de técnicas comuns de ofuscação incluem [Bruschi et al. 2007]: *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.

Essa capacidade de mutação do código original também é conhecida como metamorfismo de código [Borello e Mé 2008]. As versões metamórficas de um *malware* são geradas automaticamente por um componente do código (*engine* de metamorfismo) que é incorporado no próprio *malware* e tem a função de executar as alterações no código à medida que novas cópias do *malware* são produzidas e propagadas. Assim, mesmo pequenas alterações no código malicioso podem conduzir a falhas no processo de detecção, o que requer constantes atualizações nas bases de assinaturas. Como o número de versões metamórficas pode crescer exponencialmente, torna-se praticamente impossível sua detecção com base no modelo tradicional de assinaturas.

Diversas abordagens têm sido propostas para lidar com este problema como: a criação de um padrão de assinatura capaz de identificar grupos de códigos através de uma única sequência de identificação [Griffin et al. 2009]; a utilização de autômatos finitos para modelar chamadas de sistemas associadas ao comportamento de códigos maliciosos [Jacob et al. 2009]; a normalização do código e o levantamento do fluxo para reversão e identificação de códigos suspeitos [Cozzolino et al. 2012]; e a utilização de grafos para modelar o uso de funções [Hu et al. 2009] ou a relação de dependência entre instruções do código [Kim e Moon 2010]. Alguns dos problemas enfrentados por estas abordagens são: a criação manual dos modelos de detecção, a quantidade de informações tratadas ou ainda a alta variância nos resultados de identificação.

Este trabalho propõe uma nova metodologia para identificação de *malwares* metamórficos baseada na análise de grafos de dependência, gerados automaticamente a partir da análise de programas executáveis. A metodologia proposta é capaz de identificar as partes relevantes deste grafo, baseando-se nas características estruturais de seus vértices e arestas. Isto permite a criação de um processo mais eficiente de redução de grafos, o que diminui a quantidade de informação necessária para identificar um *malware* metamórfico. Avaliações feitas com coleções de dados reais obtidas a partir de

versões metamórficas do vírus W32.Evol[2] e W32.Polip[3] demonstram melhorias na detecção de códigos metamórficos e diminuição na variância dos resultados quando comparados com a abordagem proposta por Kim e Moon [Kim e Moon 2010], que foi usada como modelo de referência neste texto.

O restante deste artigo está organizado da seguinte forma. A seção 2 fornece alguns trabalhos relacionados ao problema de detecção de *malware* metamórfico. A seção 3 define grafos de dependência e como eles podem ser aplicados para identificar as semelhanças de códigos. A seção 4 detalha a metodologia de identificação proposta. A seção 5 fornece resultados experimentais e discussões. Finalmente, a seção 6 apresenta as conclusões e dá indicações para trabalhos futuros.

2. Trabalhos Relacionados

Abordagens alternativas para o modelo tradicional de assinaturas procuram se basear em modelos de identificação que sejam mais resistentes às técnicas de ofuscação de código empregadas por desenvolvedores de *malware* metamórficos.

Uma das propostas usa 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*, conhecidas como "*assinaturas string*", baseado na probabilidade de um símbolo vir após uma sequência qualquer de símbolos [Griffin et al. 2009]. 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 também pode ser empregada para identificar o comportamento apresentado por códigos maliciosos [Jacob et al. 2009]. Um autômato finito é usado para modelar uma sequência de chamadas a funções do sistema operacional. Em seguida, este autômato deve ser comparado com outros, previamente construídos, que modelam o comportamento apresentado por um *malware*. Entretanto, a necessidade da criação manual destes autômatos 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.

Outra alternativa combina a normalização e o mapeamento do fluxo de execução dos programas analisados [Cozzolino et al. 2012]. Primeiramente, o processo de normalização deve restaurar o código o mais próximo possível ao seu estado original. Em seguida, este código é dividido em blocos funcionais, delimitados por instruções de desvio de fluxo, e reduzidos a marcadores inteiros simples. Cada marcador é combinado com aqueles correspondentes aos dois possíveis destinos a partir deste bloco, formando uma identificação composta. Ao final, este conjunto de marcadores compostos é comparado com aqueles previamente armazenados em uma base de dados para a geração de uma pontuação que será utilizada para determinar se o código analisado se trata ou não de um *malware*. A principal limitação desta metodologia está associada ao processo de normalização, pois se este processo não obtiver os resultados esperados a

[2] http://www.symantec.com/ security_response/writeup.jsp?docid=2000-122010-0045-99

[3] http://www.symantec.com/ security_response/writeup.jsp?docid=2006-042309-1842-99

pontuação final gerada no processo de comparação será muito baixa, impedindo a correta identificação do *malware.*

Uma metodologia baseada em grafos [Hu et al. 2009] propõe a análise de códigos executáveis para a construção de uma estrutura que modele as chamadas de função presentes no código. No grafo gerado a partir desta análise, 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. Este grafo é comparado com uma base de grafos previamente existente, permitindo que a identificação do *malware* ocorra. A maior limitação deste processo está associada à dificuldade de mapear corretamente as funções criadas diretamente no código do programa, tarefa esta que pode ser ainda mais dificultada dependendo da quantidade de técnicas metamórficas aplicadas a este código.

Outro exemplo da utilização de grafos é uma proposta para detecção de códigos maliciosos inseridos dentro de *scripts* [Kim e Moon 2010]. 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. Este grafo passa por um processo de normalização que visa eliminar instruções inseridas pelas ações das técnicas de ofuscação de código, além de diminuir o tamanho do grafo original. 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 execução é bem alto, o que exige a utilização de heurísticas para a diminuição do tempo de processamento.

3. Grafos de Dependência na Identificação de Códigos Metamórficos

Grafos de dependência são grafos direcionados que representam relações de dependência entre elementos pertencentes a uma mesma estrutura [Ferrante et al. 1987]. Originalmente empregados na identificação de plágios [Liu et al. 2006], os grafos de dependência também tem sido usado como estrutura base para eliminar as ações das técnicas de ofuscação em códigos maliciosos [Kim e Moon 2010]. Nessa abordagem, cada instrução do código corresponde a um vértice e, para cada variável que esta instrução manipular, uma aresta orientada será inserida, ligando esta instrução à próxima linha de código que manipular esta mesma variável. Este trabalho propõe o uso de grafos de dependência aplicada a códigos executáveis. A Figura 1 apresenta um exemplo de grafo de dependência gerado a partir de um código *assembly*. Este procedimento é aplicado tanto a um programa suspeito de contaminação como no *malware* que será investigado, gerando dois grafos que serão então reduzidos, para eliminação de componentes derivados do metamorfismo (Figura 1.c), e finalmente comparados, para identificação de similaridades. Encontrar correspondências entre os vértices de dois grafos dados recai no problema conhecido como *Isomorfismo entre Grafos* [Garey e Johnson 1979].

Baseado nestes conceitos, este trabalho utiliza 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*.

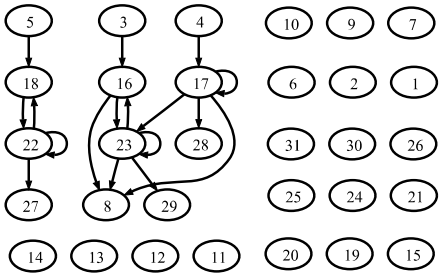
4. Metodologia de Identificação Utilizada

O processo de identificação de códigos executáveis metamórficos é composto por quatro etapas principais:

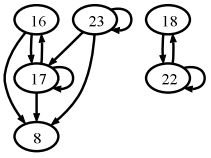
1. Reconstrução de código *assembly*, onde o programa executável passa por um processo de engenharia reversa para a obtenção do seu código equivalente em linguagem *assembly*. Esta etapa pode ser executada com o auxílio de programas como OllyDbg [4] e IDA Pro [5].
2. Geração do grafo de dependência, onde o programa gerado na etapa anterior é analisado e então usado como base para a geração do grafo de dependência.
3. 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.
4. Comparação do grafo reduzido com a base de referência, onde o grafo reduzido é comparado com um grafo correspondente a um *malware* previamente analisado.

```
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 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.

4.1. Geração dos Grafos de Dependência

O código *assembly* gerado na primeira etapa da metodologia 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.

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

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

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 tomada: *a)* caso o elemento esteja sendo manipulado pela primeira vez, o vértice correspondente àquela instrução é marcado como origem para futuras manipulações daquele mesmo elemento; *b)* caso 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; *c)* se 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, este processo 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, uma vez que todos os controles são implementados através de instruções de desvio do tipo *"jump"*. 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 e depois do trecho ignorado (`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 com 2^n folhas, onde n representa a quantidade de instruções de desvio condicional presentes no código.

4.2. Redução dos Grafos de Dependência

Reduzir um grafo de dependência significa eliminar os vértices que são considerados desnecessários, tais como vértices que representam a declaração de variáveis ou que representam trechos do código que nunca serão atingidos. Ao fim do processo de redução obtém-se um grafo que representa o comportamento principal do código, como o exemplo mostrado na Figura 1.c, que representa o resultado da redução aplicada ao grafo mostrado na Figura 1.b.

Para realizar a redução dos grafos de dependência foram definidas quatro situações onde os vértices devem ser eliminados [Kim e Moon 2010]: 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.

4.3 Identificando Elementos Relevantes do Grafo

Na metodologia proposta por Kim e Moon [Kim e Moon 2010], os grafos de dependência, gerados com base em códigos maliciosos metamórficos previamente identificados, são armazenados após o processo de redução e utilizados diretamente como base de referência para o processo de identificação de outras versões metamórficas destes mesmos *malware*, sem que qualquer informação a respeito da natureza e função dos vértices seja levada em consideração durante todas as etapas restantes da metodologia. Na proposta apresentada neste artigo, um tipo específico de vértice, denominado de *vértice de decisão*, derivado das operações de comparação entre o conteúdo de registradores, é tratado para se chegar a uma versão ainda mais reduzida do grafo de dependência.

Os vértices de decisão são gerados a partir de instruções `CMP` que são executadas antes de qualquer instrução de *jump* condicional. Como programas em linguagem *assembly* não possuem estruturas de controle de alto nível, como `if-then-else` e `do-while`, esta instrução é a principal ferramenta para a implementação das estruturas de decisão e controle de fluxo do programa. Como estas instruções não alteram o conteúdo dos registradores, os vértices gerados a partir das mesmas possuem a característica singular de não possuírem arestas que se originam a partir destes vértices.

A Figura 2, gerada a partir de um trecho de código do vírus W32.Evol, ilustra um grafo de dependência onde os vértices de decisão estão em destaque. Como as arestas representam as relações de dependência entre as instruções, quanto maior for a quantidade de arestas incidindo neste tipo de vértice, maior será sua importância para a implementação da lógica básica do programa modelado. É comum também surgir mais de um componente conexo nestes grafos, derivados de elementos de controle que não são relevantes para o processo de identificação. Por exemplo, na Figura 1.c, o componente formado pelas arestas 18 e 22 é derivado da manipulação de um contador, podendo ser desconsideradas no processo de identificação. A Figura 3 ilustra um grafo de dependência, gerado a partir de um *malware* real, onde podem ser identificados mais de um componente conexo.

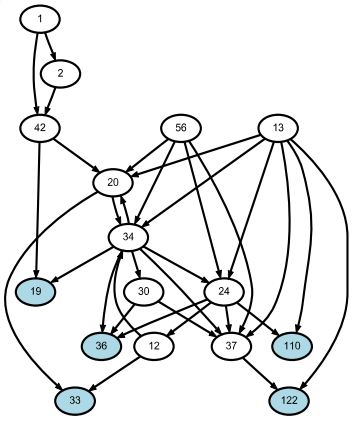


Figura 2. Grafo de dependência reduzido, com os vértices de decisão em destaque.

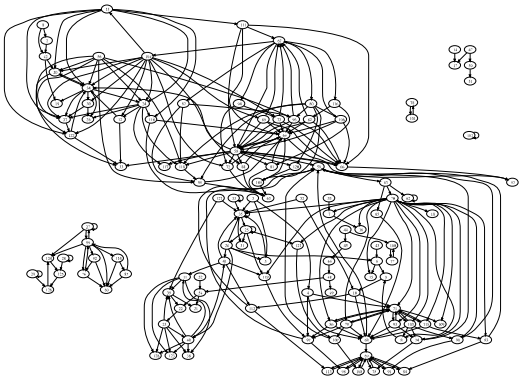


Figura 3. Grafo de dependência do *malware* W32.Evol.

Para identificar os vértices de decisão mais relevantes para o programa modelado, foi utilizado um processo dividido em quatro 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.

4.3.1 Cálculo da Menor Distância Relativa entre Vértices de Decisão

Tradicionalmente, a utilização de um algoritmo como Floyd-Warshall [Floyd 1962] [Warshall 1962] poderia ser empregada para a obtenção da distância entre cada vértice de um grafo. Entretanto, como os vértices de decisão 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 seria sempre infinita.

Assim, no momento do cálculo da distância entre cada vértice de decisão, esta metodologia considera 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 4 ilustra as arestas virtuais (destacadas em vermelho) criadas para o grafo apresentado na Figura 2.

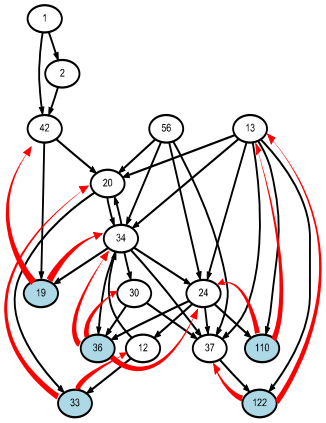


Figura 4. Arestas virtuais adicionadas ao grafo de dependência da Figura 2.

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 observando dois pontos 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. O algoritmo modificado é apresentado no Algoritmo 1.

Algoritmo 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))
```


4.3.2 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 4, a matriz de adjacências apresentada na Figura 5 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 5. Matriz de adjacências correspondente ao grafo virtual.

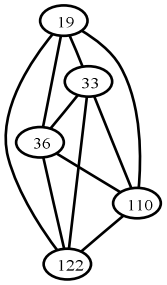


Figura 6 Clique para o grafo virtual na Figura 4.

4.3.2. Cálculo da Clique Máxima 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 da clique máxima [Konc e Janezic 2007] ao grafo virtual gerado na etapa anterior. A clique virtual, gerado a partir da matriz de adjacências da Figura 5, é apresentado na Figura 6.

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

4.3.4 Redução final do Grafo de Dependência

Na etapa final do processo de redução do grafo de dependência, a lista de vértices pertencentes à clique virtual é usada para determinar se os vértices e arestas presentes no grafo 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 na clique do grafo virtual. Caso ele não possua este caminho, este vértice e todas as arestas associadas devem ser eliminados. Um exemplo do resultado deste processo é ilustrado na Figura 7, que apresenta a versão final do grafo de dependência reduzido do *malware* W32.Evol apresentado na Figura 3. Esta nova versão reduzida será usada para identificação de versões metamórficas deste mesmo *malware*.

4.4. 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

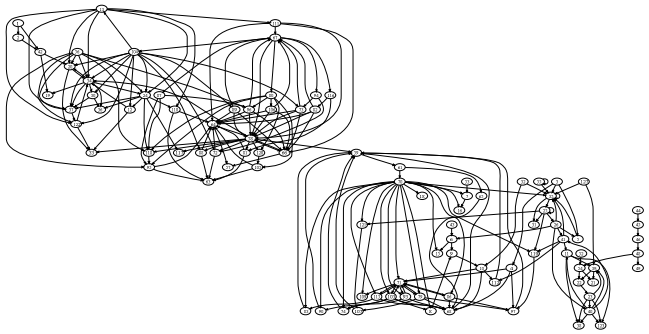


Figura 7. Versão final do grafo de dependência reduzido do W32.Evol.

problema de isomorfismo de grafos. Para os testes da metodologia foram utilizados os resultados obtidos através de três algoritmos distintos. O primeiro é um algoritmo genético tradicional, com tamanho da população constante de 100 elementos, onde 20 novos elementos, gerados com a taxa de *crossover* de 90% e uma taxa de mutação de 20%, substituem os 20 piores cromossomos existentes na geração anterior. Este processo é repetido 2000 vezes antes que o resultado final seja registrado. Os outros são implementações de duas heurísticas propostas por Kim e Moon [Kim e Moon 2010], mas com os seus resultados tratados de forma independente. A fórmula para realizar o cálculo da diferença entre os grafos G_1 e G_2 é definida na Tabela 1.

Os resultados gerados pela execução de cada algoritmo são apresentados através do cálculo da similaridade entre os grafos comparados, gerando uma pontuação de similaridade (ver Tabela 1). A menor dentre as três pontuações geradas, é então utilizada como resultado final da comparação entre G_1 e G_2, passando a ser reconhecida com a pontuação de similaridade entre os grafos comparados. Esta pontuação é usada para determinar se existe ou não contaminação por *malware* no programa analisado.

Tabela 1. Equações para realizar o cálculo da similaridade entre dois grafos

Descrição	Equação
Definições iniciais	$G_1 = (V_1, E_1), G_2 = (V_2, E_2)$ e $\lvert V_1 \rvert < \lvert V_2 \rvert$
Função de busca de uma aresta *e* em um conjunto de arestas *E*	$I(e,E) = \begin{cases} 0, & se \quad e \in E \\ 1, & caso \quad contrário \end{cases}$
Cálculo da similaridade entre G1 e G2	$similaridade(G_1, G_2) = \frac{\sum_{e \in E_1} I(e, E_2) + \sum_{e \in E_2} I(e, E_1)}{\lvert E_1 \rvert}$

5. Avaliação e Resultados Experimentais

Para avaliar a metodologia proposta, foram utilizadas versões metamórficas dos *malware* W32.Evol e W32.Polip. As amostras metamórficas do W32.Evol foram as mesmas utilizadas por Cozzolino et al. [Cozzolino et al. 2012]. No caso do W32.Polip, as amostras foram coletadas em uma base pública [Offensive Computing 2013].

5.1 Coeficiente de Ajuste da Pontuação

Para a definição de um limite máximo da pontuação de similaridade que seria utilizado como o limiar de identificação, foi criada uma base de grafos sintéticos gerados a partir de uma metodologia usada na avaliação de algoritmos para detecção de subgrafos [Conte et al. 2007]. Com base nos resultados obtidos, foi definida uma média de 25% de

similaridade como um nível mínimo para a indicação de contaminação, ou seja, a pontuação do cálculo de diferença observada entre os pares de grafos não deveria ultrapassar o limiar de 0,8 pontos. Entretanto, quando o vírus W32.Evol foi inicialmente avaliado, todos os resultados de pontuação foram superiores a 0,95, o que é muito maior que os 0,8 pontos esperados.

Após analisar tanto os conjunto de grafos sintéticos quanto os grafos gerados com base no W32.Evol, foi identificado que na base sintética a quantidade de vértices de cada par de amostras era sempre a mesma e no caso dos grafos do W32.Evol, a quantidade de vértices variou de 27,45% a 155,56% maior quando comparada com a versão do grafo definido com base de comparação do *malware*, o que explicou a diferença na pontuação obtida inicialmente.

Para tratar este problema, foi necessário aplicar um coeficiente de ajuste da pontuação de similaridade que levasse em consideração a diferença na quantidade de vértices entre os grafos comparados. Após o levantamento dessas diferenças observou-se que, em média, os grafos associados às versões metamórficas possuíam 1,67 vértices a mais que o grafo associado ao código original. Este valor foi então aplicado aos resultados iniciais como um coeficiente redutor, segundo a fórmula $PSA = PSO/r$, onde "*PSA*" é a pontuação de similaridade ajustada, "*PSO*" é a pontuação de similaridade original e "*r*" representa o coeficiente de redução. Esta nova sequência de valores ficou abaixo da pontuação definida como limiar para identificação do *malware* procurado. O resultado deste ajuste é ilustrado na Figura 8. Todos os dados apresentados nos próximos gráficos têm o coeficiente de redução correspondente já aplicado.

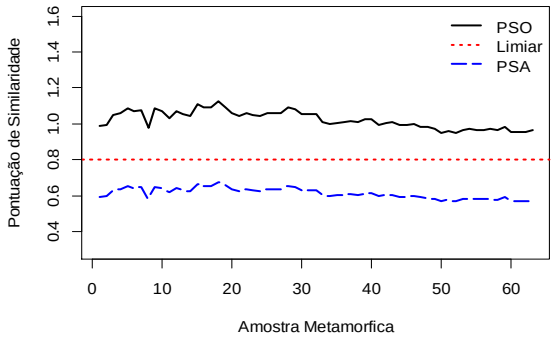


Figura 8. Utilizando o coeficiente de redução na pontuação de similaridade de referência para W32.Evol.

5.2 Resultados com o Processo de Redução Aprimorado

O primeiro conjunto de testes foi baseado no *malware* W32.Evol. Inicialmente, um grafo base para comparação foi gerado a partir de uma versão ainda sem alterações metamórficas do W32.Evol. A seguir, as 63 amostras metamórficas disponíveis foram comparadas diretamente com este grafo base. A pontuação obtida apresentou um coeficiente de variação (desvio padrão dividido pela média) de 4,65%. Testes realizados sobre o mesmo conjunto de elementos, avaliados por outro método [Cozzolino et al. 2012], obtiveram um coeficiente de variação de 47,29%.

Para avaliar a metodologia de redução aprimorada proposta, o arquivo contendo grafo de dependência usado como base de comparação foi então submetido a todo o processo descrito na seção 4.3. Como resultado, a quantidade de vértices do grafo base caiu em 23,52%, passando de 153 para apenas 117 vértices. Finalmente, depois de coletados os resultados da comparação deste novo arquivo reduzido com as 63 versões metamórficas, foram obtidos os resultados apresentados na Figura 9.a, onde são comparados com os resultados da metodologia de referência [Kim e Moon 2010], cujo processo de redução empregado é apenas aquele descrito na seção 4.2. Os resultados obtidos tanto pelo uso da metodologia de referência como pelo uso do processo de redução aprimorado, obtiveram sucesso na identificação de todas as amostras metamórficas, atingindo coeficientes de similaridade abaixo do limiar de identificação definido.

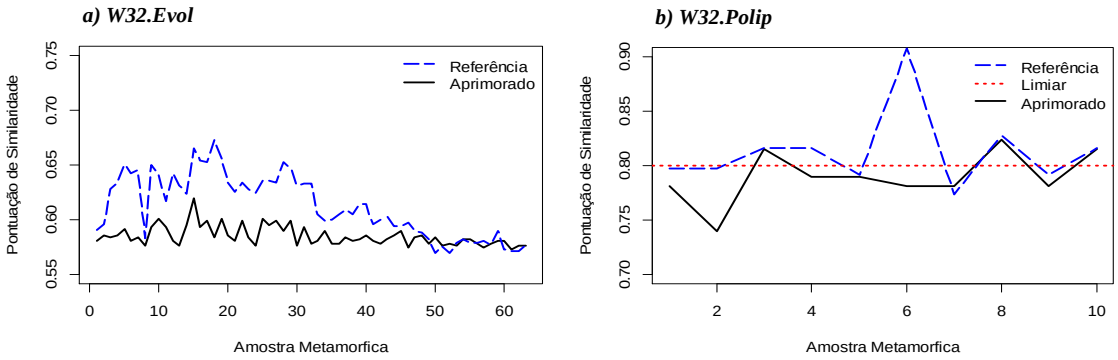


Figura 9. Comparação entre os resultados da metodologia de referência e aqueles obtidos pelo processo de redução aprimorado para os códigos maliciosos W32.Evol (a) e W32.Polip (b).

Entretanto, os resultados obtidos com o uso do processo de redução aprimorado possuem um coeficiente de variação de apenas 1,51%, demonstrando que este processo manteve os componentes do grafo base associado aos elementos mais relevantes do programa a partir do qual foi gerado, diminuindo consideravelmente a interferência que as estruturas eliminadas do grafo de dependência introduziram nos resultados obtidos através da metodologia de referência.

Um segundo conjunto de testes foi realizado com onze amostras do *malware* metamórfico W32.Polip. A quantidade de amostras disponíveis não era muito grande, mas foi possível selecionar uma delas como elemento de referência para a comparação com as demais amostras. Uma dificuldade específica neste conjunto de dados foi o cálculo do coeficiente de redução. A diferença na quantidade de vértices entre o grafo de referência e algumas amostras era muito grande. Uma das amostras, por exemplo, possuía mais de quatro mil vértices, enquanto o grafo de referência possuía apenas 23 vértices. Se estas amostras entrassem no cálculo, o valor gerado seria um coeficiente de redução superior a 20, o que distorceria os resultados. Assim, duas amostras que possuíam mais de dois mil vértices foram desconsideradas, deixando como coeficiente de redução final o valor de 1,19 para os resultados de comparação do W32.Polip.

Os resultados finais da avaliação do W32.Polip são apresentados na Figura 9.b. Neste conjunto de dados a metodologia de referência identificou apenas 50% das amostras metamórficas, enquanto que o uso do processo de redução aprimorado

permitiu a identificação de 70% destas mesmas amostras. Novamente tivemos uma melhoria na estabilidade dos resultados produzidos, apesar de não se ter observado uma diferença tão significativa quanto no primeiro conjunto de testes, já que a o coeficiente de variação correspondente à metodologia de referência foi de 4,51%, contra um coeficiente de variação de 3,05% para a metodologia de redução aprimorada.

6. Conclusões

Neste trabalho foi proposta uma abordagem destinada a identificar *malware* metamórfico através da comparação de grafos de dependência armazenados em uma base de referência, onde foi aplicado um processo de redução aprimorado baseado na diferenciação de nós em conjunto com a utilização de estruturas virtuais. Embora os dados experimentais com a base sintética de teste apresentaram características que não puderam ser observadas nos dados iniciais obtidos da avaliação de programas reais, a utilização de um coeficiente de redução, diretamente relacionado com a diferença entre o número de vértices dos grafos avaliados, tornou a pontuação consistente com o modelo de comportamento esperado.

A utilização de estruturas temporárias, como as arestas virtuais e a geração da clique virtual, forneceu uma maneira gerenciável de lidar com as características dos grafos de dependência, mapeando a relação estrutural entre os componentes mais relevantes dos grafos. Isto permitiu que o processo de redução aprimorado diminuísse o tamanho dos grafos de dependência da base de referência, sem prejuízo para sua utilização no processo de identificação. Nos testes com os *malware* W32.Evol e o W32.Polip, houve diminuição na variância dos resultados e, no caso do W32.Polip também foi observada uma melhoria nos resultados de identificação. Assim, os resultados obtidos em todos os testes mostram o potencial da abordagem proposta na melhoria do processo de identificação de código metamórfico.

As próximas etapas serão dedicadas à modificação do cálculo de pontuação de similaridade, eliminando a necessidade do coeficiente de redução, e o desenvolvimento de uma nova abordagem para resolver o problema do isomorfismo máximo de subgrafo entre grafos de dependência, com foco na redução do tempo de processamento necessário. Esta nova abordagem deve aproveitar ao máximo as informações sobre o tipo de vértice, em conjunto com o uso de ordenação topológica, já que o primeiro permite um processo de comparação seletivo, que manipule apenas elementos de mesma natureza e o último permite organizar a estrutura do grafo de uma forma que está mais relacionado com a ordem de execução das instruções do programa original.

Agradecimentos

Este trabalho foi financiado pela Fundação de Amparo à Pesquisa do Estado do Amazonas (FAPEAM) através do processo 062.03178/2012 (Edital Universal Amazonas). Agradecemos ainda a FAPEAM e a CAPES pelo apoio financeiro com bolsa de doutorado.

Referências

Baker, W. Hutton, A. Hylender, C. D. Pamula, J. Porter, C. e Spitler, M. (2011). 2011 data breach investigations report. Verizon RISK Team. http://www. verizonbusiness.com/resources/ reports/rp_databreach-investigations-report-2011_en_xg.pdf.

Bomze, I. M. Budinich, M. Paradalos, P. M. e Pelillo, M. (1999). “The maximum clique problem”. Handbook of combinatorial optimization. Springer US, p. 1-74.

Borello, J. e Mé, L. (2008). “Code obfuscation techniques for metamorphic viruses”, Journal in Computer Virology, vol. 4, núm. 3, pág. 211-220.

Bruschi, D. Martignoni, L. e Monga, M. (2007). "Code Normalization for Self-Mutating Malware" , IEEE Security & Privacy, v. 5, n. 2, p. 46-54.

Conte, D. Foggia, P. e Vento, M. (2007). “Challenging Complexity of Maximum Common Subgraph Detection Algorithms: A Performance Analysis of Three Algorithms on a Wide Database of Graphs”,J. Graph Algorithms Appl, v. 11,n. 1, p. 99-143.

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.

Ferrante, J. Ottenstein, K. J. e Warren, J. D. (1987). “The program dependence graph and its use in optimization”, ACM Transactions on Programming Languages and Systems (TOPLAS), v. 9, n. 3, p. 319-349.

Floyd, R. (1962). “Algorithm 97: shortest path”. Communications of the ACM 5.6, 5:345.

Garey, M. R. e Johnson, D. (1979). “Computers and Intractability: A Guide to the Theory of NP-Completeness”, W. H. Freeman & Co.

Griffin, K. Schneider, S. Hu, X. e Chiueh, T. C. (2009). “Automatic Generation of String Signatures for Malware Detection”. Proceedings of the 12th Symposium on Recent Advances in Intrusion Detection (RAID2009), Brittany, França, p. 101-120.

Hsiao, S.-W. Sun, Y. S. Chen, M. C. e Zhang, H. (2010). "Behavior Profiling for Robust Anomaly Detection," IEEE International Conference on Wireless Communications, Networking and Information Security, p. 465-471.

Hu, X. Chiueh, T.-c. e Shin, KG. (2009). “Large-scale malware indexing using function-call graphs”. Proceedings of the 16th ACM conference on Computer and communications security, p. 611-620.

Jacob, G. Debar, H. e Filiol, E. (2009). “Malware Detection using Attribute-Automata to parse Abstract Behavioral Descriptions”, CoRR, abs/0902.0322.

Karin, A. (2006). “Automatic Malware Signature Generation”. 16 de Outubro, 2006. Disponível em http://web.it.kth.se/~cschulte/teaching/theses/ICT-ECS-2006-122.pdf.

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

Konc, J. e Janezic, D. (2007). “An improved branch and bound algorithm for the maximum clique problem”. Communications in Mathematical and in Computer Chemistry, n. 58, p. 569-590.

Liu, C. Chen, C. Han, J. e Yu, P. S. (2006). “GPLAG: detection of software plagiarism by program dependence graph analysis”. Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, p. 872-881.

Moura, A. V. e Rebiha, R. (2009). “Automated Malware Invariant Generation”, International Conference on Forensic Computer Science (ICoFCS), p. 7.

Notoatmodjo, G. (2010). “Detection of Self-Mutating Computer Viruses”, Disponível em http://www.cs.auckland.ac.nz/compsci725s2c/archive/termpapers/gnotoadmojo.pdf, Department of Computer Science, University of Auckland, New Zealand.

Offensive Computing. (2013). “Offensive Computing”. http://www.offensivecomputing.net/. Nov. 2013.

Warshall, S. (1962). “A theorem on boolean matrices”. Journal of the ACM (JACM), n. 9, p. 11-12.

