UNIVERSIDADE FEDERAL DO AMAZONAS
PRO REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
DEPARTAMENTO DE APOIO À PESQUISA
PROGRAMA INSTITUCIONAL DE INICIAÇÃO CIENTÍFICA

Bolsista: Leonardo Alexandre Lima de Sales

UTILIZAÇÃO DE GRAFOS DE DEPENDÊNCIA PARA DETECÇÃO DE *MALWARE* METAMÓRFICOS

Manaus
2013

UNIVERSIDADE FEDERAL DO AMAZONAS
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
DEPARTAMENTO DE APOIO À PESQUISA
PROGRAMA INSTITUCIONAL DE INICIAÇÃO CIENTÍFICA

RELATÓRIO FINAL
PIB-E-0200/2012

UTILIZAÇÃO DE GRAFOS DE DEPENDÊNCIA PARA DETECÇÃO DE *MALWARE* METAMÓRFICOS

Bolsista: Leonardo Alexandre Lima de Sales

Orientador: Prof. Dr. Eduardo James Pereira Souto

MANAUS
2013

RESUMO

O termo *malware* (*malicious code)* ou código malicioso é usado para classificar um software destinado a se infiltrar em um sistema de computador alheio sem consentimento e conhecimento do usuário, com o intuito de causar algum dano ou roubo de informações. A maioria das fraudes ocorre nos sistemas financeiros e órgãos governamentais por se tratar de sistemas que contém forte esquema de segurança e guardar informações sigilosas sobre movimentações financeiras de diversas organizações.

Atualmente, a evolução dos códigos maliciosos tem ocorrido de forma rápida, muitos atacantes empregam técnicas de ofuscação de códigos para ludibriar os sistemas de detecção de atividades maliciosas, introduzindo alterações na codificação de forma automatizada a cada nova cópia produzida. Tal abordagem, mantém a funcionalidade do código, mas inviabiliza sua identificação pelas abordagens tradicionais de detecção. Esta característica de mudança contínua também é conhecida como “metamorfismo”.

Este projeto de pesquisa apresenta um estudo sobre o uso de grafos de dependência na modelagem de códigos metamórficos. O objetivo é reconhecer o comportamento de um programa malicioso a partir das alterações realizadas pelas operações metamórficas. Os resultados mostram que o grafo de dependência permite modelar o comportamento e se torna uma estrutura mais resistente às técnicas de ofuscação de detecção presentes no código dos sistemas infectados.

Um estudo foi conduzido usando 06 (seis) gerações de códigos gerados a partir da evolução metamórfica do vírus de computador Evol W32. Os resultados mostram que o grafo de dependência permite modelar o comportamento, sendo capaz de detectar todas as versões de código metamórficos usada nos experimentos.

Palavras chave: código malicioso, *malware*, metamorfismo.

SUMÁRIO

1 Introdução 5
2 Revisão Bibliográfica 6
2.1 Detecção por Assinaturas 6
2.2 Ofuscação de código 6
2.3 Trabalhos Relacionados 7
3 Metodologia 8
3.1 Visão Geral 8
3.2 Engenharia Reversa 9
3.3 Pré-processamento 9
3.4 Grafos de Dependência 10
3.5 Redução do Grafo de Dependência 10
3.6 Seleção de Características Relevantes 11
4 Resultados Experimentais 12
4.1 Pré-Processamento 12
4.2 Grafo de Dependência 13
4.3 Estudo de Caso 15
5 Conclusão e Trabalhos Futuros 19
6 Referências 20
7 Cronograma 22

1 Introdução

Desde a popularização da Internet, em meados dos anos 90, a propagação de softwares maliciosos (*malware*) tem aumentado significativamente. *Malware* é um termo derivado da expressão "*Malicious Software*" que classifica todos os tipos de softwares maliciosos que causam perdas, roubo ou vazamento de dados confidenciais (Skoudis, 2004). Estes programas maliciosos são utilizados para atacar sistemas computacionais em geral, desde computadores pessoais até grandes redes corporativas e possuem diferentes finalidades, dependendo da intenção que motivou a invasão, variando desde ganho financeiro e bloqueio de atividades empresariais a roubo de dados confidenciais.

Atualmente, a principal técnica de detecção de *malware* e amplamente utilizada em ferramentas comerciais é baseada na busca por assinaturas (Karin, 2006). Ela consiste na procura de uma sequência de *bytes* (assinatura) armazenada em uma base de dados de assinaturas que caracterizam o *malware*. Na tentativa de dificultar a identificação de suas criações, os escritores desses códigos maliciosos têm empregado técnicas para evitar que a busca por assinaturas tenha sucesso. Uma dessas técnicas possibilita a alteração do código do vírus à medida que sua propagação ocorre. Tal técnica é conhecida como "metamorfismo". As versões metamórficas de um *malware* são geralmente produzidas automaticamente por um componente do código (*engines* de metamorfismo) que é incorporado no próprio *malware*. Assim, mesmo pequenas alterações no código viral podem conduzir a falhas no processo de detecção ou requerer constantes atualizações na base de assinaturas para inserir as variantes recém-criadas. Como o número de versões metamórficas pode crescer exponencialmente, torna-se praticamente impossível sua detecção usando apenas assinaturas.

Neste contexto, este projeto de pesquisa apresenta uma metodologia para identificação de *malware*. O objetivo é criar um modelo que represente as relações de dependência entre as instruções que constituem um programa e os dados que as instruções manipulam, representando essa relação em uma estrutura conhecida como Grafo de Dependência. Assim, ao invés de utilizar o processo de comparação de trechos de códigos, este modelo de detecção manipula e compara grafos de dependência gerados a partir de um *malware* conhecido e de um programa suspeito de contaminação. Um processo de redução, que elimina derivações do metamorfismo no código, e técnicas direcionadas a solucionar os problemas clássicos de isomorfismo entre grafos e máximo isomorfismo de subgrafo são usados como alternativa de comparação de códigos suspeitos com uma base de referência de *malware*. Esta abordagem representa uma melhoria no processo de identificação, pois utiliza Grafos de Dependência como mecanismo de representação do *malware.* Tal representação tem se mostrado ser mais resistente às técnicas utilizadas para ocultar a identidade original de um programa.

2 Revisão Bibliográfica

Esta seção descreve conceitos fundamentais para compreensão deste trabalho, contemplando diferentes tipos de vírus, o modelo tradicional de detecção, as técnicas utilizadas pelos atacantes (desenvolvedores de *malware*) pra dificultar a identificação de suas criações e, finalmente, a apresentação de um conjunto de técnicas alternativas ao modelo tradicional que são direcionadas a neutralizar os esforços de ocultação da identidade de um *malware*.

2.1 Detecção por Assinaturas

Assinaturas são definidas por uma sequência de bytes extraída do corpo de um código binário específico (Karin, 2006). O processo de formação de assinaturas de um programa consiste em gerar uma assinatura baseada em partes de código do programa que não se repetem ou que caracterizam o comportamento do programa. A Tabela 1 apresenta um exemplo de trechos de códigos usados para gerar uma assinatura. Para ser efetiva, é necessário que a base de dados de assinaturas de *malware* seja atualizada constantemente. Normalmente, essa base de dados é fornecida pelo proprietário do programa antivírus. Caso uma assinatura seja encontrada em um programa infectado, o software antivírus age buscando proteger o sistema operacional contra possíveis danos.

Tabela 1: processo de geração de uma assinatura.

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

2.2 Ofuscação de código

Segundo Borello e Mé (Borello e Mé, 2008), a contínua evolução dos métodos utilizados para o desenvolvimento de vírus busca, como uma de suas metas, tornar estes programas indetectáveis. Este processo, também conhecido como “ofuscação”, busca garantir que o código binário não seja reconhecido durante a consulta a uma base de dados de assinaturas. O trabalho de Notoatmodjo (Notoatmodjo, 2005) classifica este tipo de vírus como “5a. geração”, onde a principal característica é a capacidade de infectar outros sistemas com versões modificadas de si mesmo, sendo ainda subdivididos em dois grupos: grupo de *malwares* “polimórficos”, que utilizam técnicas de criptografia combinadas com uma mudança contínua da chave criptográfica visando garantir a diferenciação contínua dos códigos gerados; grupo de *malware* “metamórficos”, que utilizam técnicas diferentes da

criptografia para gerar suas mutações. Rad e Masrom (Rad e Masrom, 2010) oferecem mais detalhes a respeito dos “vírus metamórficos”. Destaca-se ainda que a efetividade desta técnica de ocultação se baseia no fato de que a maioria das assinaturas de vírus é criada com base na varredura do código binário do vírus. Assim, para evitar a detecção, novas cópias daquele mesmo código mantêm a estrutura semântica original, incluindo alterações sintáticas que os tornam incompatíveis com a assinatura gerada originalmente.

2.3 Trabalhos Relacionados

Esta seção descreve alguns trabalhos relacionados ao processo de identificação e detecção de *malware*.

A identificação de programas gerados a partir de uma mesma *engine* metamórfica é um dos processos mais complicados de se executar, visto que existem muitas técnicas de mutação criadas para dificultar a detecção de atividades e códigos maliciosos. As mutações feitas por uma determinada *engine* buscam características como instruções, assinaturas, mecanismos sequenciais de instruções que caracterizam essa *engine* (Lakhotia e Chouchane, 2006).

O trabalho de (Bruschi, 2007) é baseado no fato de que a maioria das transformações utilizadas por *malware* visando ludibriar os sistemas de detecção tem como efeito colateral o aumento do tamanho do código binário produzido. Como o código gerado não é um código otimizado, instruções são inseridas exclusivamente para mascarar o código original. Bruschi propõe a normalização deste código visando eliminar o código “morto” inserido na versão original, facilitando o reconhecimento do *malware*.

A técnica proposta por (Zhang, 2008) consiste em uma análise estática do código fonte investigado em busca de chamadas ao sistema e, com base em uma sequência específica de chamadas (assim como o processo de análise comportamental de antivírus no processo de detecção baseado em comportamentos), cria um mecanismo que serve como referência para identificação do código, formando a base para o processo de identificação.

(Kim e Moon, 2010) apresentam um mecanismo de avaliação de códigos script buscando encontrar códigos maliciosos inseridos. Com a ideia de evitar que o polimorfismo camufle a existência de código malicioso, usa-se grafos de dependência para representar relações de dependência em código semântico focando na manipulação das variáveis declaradas e usadas para relacionar suas dependências. Após essa geração e representação, ocorre a normalização do grafo. Esse processo elimina os vértices ou blocos de vértices que não possuem relação com outros vértices que pertencem ao corpo do grafo primário. A partir desse ponto, o grande desafio é encontrar o máximo isomorfismo entre o grafo normalizado de um código em análise e o grafo de dependência que modela o subgrafo.

3 Metodologia

Este capítulo apresenta a metodologia para a geração do grafo de dependência, bem como o processo de redução de grafos para minimização da ação das técnicas de ofuscação empregadas nos códigos metamórficos. A metodologia deste trabalho pertence à família de normalização de código, que propõe a atividade de pré-processamento, geração de grafos de dependência e redução de grafos de dependência, antes de proceder a etapa de análise comportamental. O processo de detecção do metamorfismo é composto basicamente pela fase de geração e redução do grafo de dependência, além da análise comportamental do grafo para detectar as semelhanças entre os grafos, ou seja, entre o comportamento de cada código.

3.1 Visão Geral

A metodologia visa analisar um padrão comportamental existente entre os códigos em análise. Sendo assim, é necessário criar grafos de dependência reduzidos. Estes grafos reduzidos representam o padrão comportamental do código e o processo de redução elimina as técnicas de ofuscação inseridas nos códigos metamórficos. Para alcançar a etapa de análise do comportamento dos códigos em análise é necessário executar as 05 (cinco) etapas seguintes:

- Etapa 1 – *Engenharia Reversa* – Inicia com o processo de reversão do código binário executável em linguagem *assembly*, que possibilitará a interpretação das instruções;
- Etapa 2 – Pré-processamento – Nesta etapa são levantadas as instruções de desvio de fluxo com finalidade de criar identificadores no código. Isto é feito para atender as necessidades da detecção de técnicas de ofuscação existentes, visto que as estruturas de desvio de fluxo são também empregadas como alternativa desse tipo de técnica.
- Etapa 3 – Geração do Grafo de Dependência – O grafo de dependência modela o comportamento do código pré-processado e ajuda na identificação de uma sequência de instruções que manipulam a mesma informação específica. Mesmo com existência de técnicas de ofuscação, o grafo de dependência permanece o mesmo a cada geração ou existência dessas técnicas.
- Etapa 4 – Redução do Grafo de Dependência – Nesta fase, o grafo de dependência é reduzido com intenção de remover estruturas irrelevantes que são modeladas como técnicas de ofuscação que são detectáveis no processo de redução. Assim, a partir deste ponto, o grafo de dependência reduzido é gerado e serve como base para a análise do padrão comportamental do *malware;*

- Etapa 5 – Seleção de características relevantes – Ao término do processo de redução, a seleção de características relevantes pode ser executada em busca de um padrão comportamental ou até mesmo comparação e análise dos grafos gerados. Essa etapa busca identificar semelhanças entre a estrutura gerada e um grafo de dependência gerado do código inicial, antes de aplicar as técnicas de geração e redução de grafos de dependência, visando elencar semelhanças entre os grafos e analisar características dos códigos em análise.

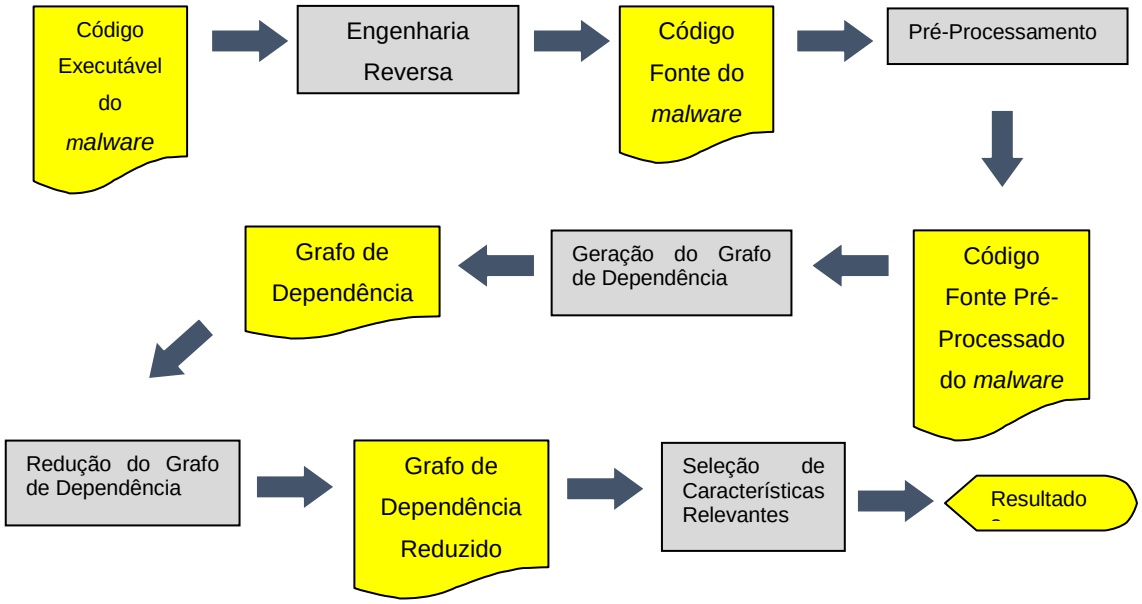


Figura 1: Representação da metodologia proposta.

3.2 Engenharia Reversa

Antes de iniciar o processo de geração dos grafos de dependência, a primeira etapa da metodologia é o processo de engenharia reversa dos códigos que serão manipulados, visto que este processo tem por finalidade criar identificadores que facilitam as buscas por instruções no corpo do código. Este processo transforma o código binário executável em um equivalente expresso em términos de linguagem *assembler*.

3.3 Pré-processamento

O pré-processamento se inicia com o levantamento das instruções de desvio de fluxo existentes no código *assembly*. Para isso, todas as instruções que constituem o código *assembly* gerado a partir do processo de engenharia reversa do programa executável e, análisee, são analisadas para a identificação das instruções de desvio de fluxo.

O desvio de fluxo é caracterizado pelo uso de instruções do tipo *Jump*. Essas instruções são usadas para indicar ao processador desvios do fluxo natural de sequência de

instruções e, na linguagem *assembly,* são utilizadas para implementar funcionalidades de *loops*, normalmente quando indicam o desvio para uma instrução com endereço inferior ao da instrução atual, ou estruturas de seleção (se-então ou se-então-senão), quando apresentam desvios para endereços de instrução maiores que as instruções imediatamente posteriores ao endereço de instrução corrente. Além disso, o uso intensivo de instruções de desvio de fluxo é também empregado como uma das técnicas de ofuscação de código, permitindo que, as instruções que constituem o programa, sejam trocadas de ordem, sem que a sequência original de execução seja perdida. Todas estas características demandam o controle cuidadoso destas instruções de desvio de fluxo.

3.4 Grafos de Dependência

Grafos de dependência são grafos direcionados que representam relações de dependência de entre elementos pertencentes a uma mesma estrutura (Kim e Moon, 2010). Os grafos de dependência gerados durante a execução deste trabalho têm por objetivo modelar as relações de dependência que existem entre as instruções que constituem o código executável de um programa, em função dos registradores que são manipulados em cada uma destas instruções.

Essas relações de dependência mapeiam os fluxos das informações que são manipuladas pelas instruções que constituem o programa. Um caminho no grafo pode ser interpretado como uma representação da cadeia de instruções destinadas a manipular uma informação específica. Assim, mesmo com uma codificação alterada pelo uso das técnicas de ofuscação de código, as informações manipuladas e os resultados gerados a partir de suas manipulações devem ser manter, ou o código gerado não será capaz de produzir o resultado final esperado. Assim, esta estrutura se mantém muito similar àquela do programa original, independentemente da quantidade de metamorfismos presentes em uma nova versão metamórfica desde mesmo programa.

Após concluída a etapa de pré-processamento e transformado o código em um novo formato, a geração do grafo de dependência é iniciada. Ao final da geração dos grafos de dependência, é iniciado o processo de redução de grafos, onde são removidas todas as características metamórficas inseridas no código pelo processo de ofuscação causado pelos metamorfismos, eliminando vértices e arestas desnecessárias e permitindo extrair os elementos fundamentais que caracterizam o código original.

3.5 Redução do Grafo de Dependência

A técnica de redução (Kim e Moon, 2010) permite eliminar do grafo de dependência de um código em análise, os vértices e arestas gerados em função da substituição e inclusão de instruções oriundas das técnicas de metamorfismos sob as quais o código original foi

submetido. Como consequência disto, ao termino do processo de redução, o tamanho final do grafo produzido é bem menor do que o original.

Este processo também é conhecido como “normalização” e, através dele, é possível comparar os modelos de grafos buscando encontrar um padrão comportamental, visto que a estrutura se baseia nas relações de dependência entre códigos e as informações que são manipuladas. Independente da forma como foi codificada e da quantidade de instruções irrelevantes introduzidas, o comportamento continuará imutável.

Para que a redução do grafo de dependência seja possível, é necessário atender a quatro condições básicas para que vértices e arestas sejam eliminados:

1) Vértices que não possuem arestas direcionadas a ele e que possuem apenas uma aresta direcionada a outro vértice. Normalmente, esses casos acontecem por serem vértices que caracterizam declarações de variáveis;
2) Vértices que possuem apenas uma aresta direcionada a ele e que não possuem arestas direcionadas a outro vértice. Significa que o primeiro vértice usa o valor do vértice anterior;
3) Vértices que possui apenas um vértice direcionado a ele e apenas um vértice partido dele em direção a outro vértice. Significa que o vértice serve como transportador de informações do vértice anterior para o próximo vértice;
4) Vértices sem arestas direcionadas;

Caso ocorra a existência de vértices que atendam a esses 4 requisitos, o processo de redução de grafos de dependência é perfeitamente possível.

3.6 Seleção de Características Relevantes

A técnica de redução permite identificar metamorfismos inseridos, o que de fato, ao término do processo de redução de grafos, o tamanho e até mesmo formato do grafo é modificado, propondo ter o mesmo modelo ao final do processo.

Ainda relacionado ao trabalho de redução de grafos, é possível comparar os modelos de grafos para que se encontre um padrão comportamental, visto que a escolha da estrutura baseia-se no fato de modelar o comportamento do programa em análise.

Para investigar a metodologia proposta, foram coletados 63 versões de um mesmo malware. Essa escolha deve-se ao fato de poder mesclar as versões metamórficas com o código original a fim de comparar número de linhas e o grafo de dependência antes e após o processo de redução de grafos. Os resultados obtidos serão apresentados na seção 4.3.

4 Resultados Experimentais

Esta seção detalha o estudo de caso onde a metodologia proposta foi aplicada no processo de geração de grafo de dependência e análise comportamental do *malware*. O objetivo é demonstrar que através da metodologia adotada o grafo de dependência modela o comportamento do *malware* a fim de resistir às técnicas de ofuscação inseridas no código metamórfico.

4.1 Pré-Processamento

O pré-processamento se inicia com o levantamento das instruções de desvio de fluxo existentes no código. Para isso, é criada uma lista onde são armazenadas as instruções presentes no código para identificação dos desvios de fluxo.

O desvio de fluxo é caracterizado pela instrução *Jump*. Essas instruções são usadas para indicar ao processador de instruções que a próxima instrução a ser executada não é exatamente a próxima instrução que está imediatamente a seguir, mas sim, outra instrução indicada.

A Figura 1 apresenta uma descrição das funções de desvio de fluxo (jump) presentes nos códigos preliminares do Evol W32 usados nos experimentos.

1. jmp – realiza operação de Jump;
2. jz – realiza operação de Jump caso a flag Z seja igual a 1;
3. jnz – realiza operação de Jump caso a flag Z seja igual a 0;
4. jne – realiza operação de Jump caso o valor da comparação não seja;
5. jnl - realiza operação de Jump caso o valor não seja negativo;
6. jnle – realiza operação de Jump caso o valor não seja negativo ou igual;
7. jng – realiza operação de Jump caso o valor não seja positivo;
8. jnge – realiza operação de Jumo caso o valor não seja positivo ou igual;
9. jnb – realiza operação de Jumo caso o valor não seja menor;
10. jnbe – realiza operação de Jump caso o valor não seja igual ou menor;
11. jna – realiza operação de Jump caso o valor não seja maior;
12. jnae – realiza operação de Jump caso o valor não seja maior ou igual;

Figura 1: Funções de desvio de fluxo (jump) presentes no Evol W32

Depois de realizar uma busca pela ocorrência dessas instruções presentes no código, são armazenadas as linhas de código para onde essas instruções desviam. A Tabela 2 exemplifica uma instrução de desvio e o trecho de código para onde o fluxo de controle está sendo redirecionado.

Tabela 2: Instrução que manipula o fluxo causado desvio da linha 48 para linha 201.

Origem			Destino		
Linha	Identificador	Instrução	Linha	Identificador	Instrução
48	CODE:0040102B	jz loc_4011D6	201	CODE:004011D6	loc_4011D6:
			202	CODE:004011D6	add esp, 4
			203	CODE:004011D9	pop ebp
			204	CODE:004011DA	retn

Como mostrado na Tabela 2, o desvio do fluxo acontece na execução da instrução jz loc_4011D6 que desvia o fluxo da linha 48 para linha 201, onde executa a função loc_4011D6 e termina na linha 204 com a instrução retn.

Após armazenamento da linha de código onde começa a declaração da função chamada através da função de desvio, é realizada outra busca no código para que seja trocada a identificação a função, que são chamados de rótulos. Esse rótulo é substituído pela linha armazenada que indica onde começa a declaração da função. Na Tabela 3 é mostrado o resultado da mudança de rótulo, onde temos a substituição do rótulo original (loc_4011D6) no trecho de código de origem, pelo número da linha (201) onde o rótulo marcador de destino do desvio está posicionado. Esta informação será depois utilizada na etapa de geração do grafo de dependência, facilitando o processo de geração deste grafo.

Tabela 3: Representação da mudança de rótulo.

Origem			Destino		
Linha	Identificador	Instrução	Linha	Identificador	Instrução
48	CODE:0040102B	jz 201	201	CODE:004011D6	loc_4011D6:
			202	CODE:004011D6	add esp, 4
			203	CODE:004011D9	pop ebp
			204	CODE:004011DA	retn

4.2 Grafo de Dependência

A geração do grafo de dependência utiliza os índices criados ao longo do pré-processamento. O primeiro passo é a contagem da quantidade de instruções presentes no código analisado. Esta informação é usada para definir a quantidade de arestas que o grafo irá possuir. Este número é primeira informação que será parte da representação gerada do grafo gerado.

A seguir, inicia-se o processo de identificação dos registradores utilizados no programa. O processo de identificação consiste em listar as ocorrências de instruções que manipulam o conteúdo dos registradores. Para este trabalho, foram utilizadas as seguintes instruções: dec, inc, not, neg, and, mov, add, or, shl, shr, sub, xor. A cada ocorrência dessas

instruções, a próxima palavra encontrada representa um registrador. A Tabela 4 representa a ocorrência de um registrador no código

Tabela 4: Representação de uma instrução de atribuição onde o valor 0 é atribuído ao registrador eax.

Linha	Identificador	Instrução
05	CODE:004011BC	mov eax, 0

Após encontrar a existência de uma instrução que manipula o registrador, é armazenado o identificador deste registrador em uma lista dinâmica que servirá de base para o controle das instruções que tiverem alterado o conteúdo do registrador, pois estas serão os pontos de partida as arestas orientadas. A partir desse momento, a cada ocorrência de uma instrução que manipule o conteúdo deste mesmo registrador, é criada uma nova aresta ligando o vértice associado a última instrução que alterou o conteúdo daquele registrador ao vértice correspondente à próxima instrução manipular este mesmo registrador.

Uma lista que descreve a existência de um registrador e seus códigos semânticos a partir da busca pelas instruções que manipulam o conteúdo dos registradores. A Tabela 5 mostra uma saída gerada por este processo. Assim, todas as informações necessárias para construir o grafo de dependência são representadas por uma simples sequência de números inteiros.

Tabela 5: Exemplo de grafo de dependência gerado utilizando pares ordenados.

Grafo de dependência baseado em lista
(19) (3,6) (6,10) (6,13) (13,18) (4,7) (7,10) (7,13) (7,17) (13,10) (13,13) (13,18) (5,8) (8,12) (12,12) (12,16)

A Figura 2 representa a geração do grafo de dependência em função:

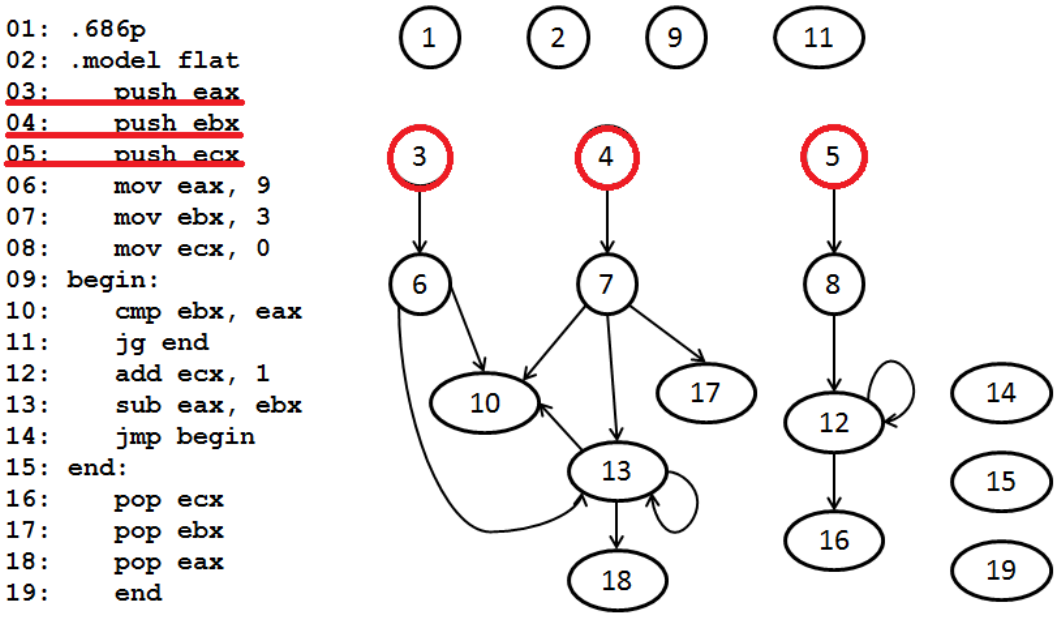


Figura 2: Representação do grafo de dependência

4.3 Estudo de Caso

Nesta seção são apresentados os resultados obtidos após o tratamento e avaliação de uma base de arquivos em que consiste de seis (06) gerações de mutações do *malware* Evol W32, totalizando 63 arquivos. Estes arquivos foram tratados e processados através da metodologia proposta e inseridos no processo de análise da evolução do metamorfismo através das gerações do *malware* que identifica os comportamentos metamórficos utilizando grafos de dependência para detecção do *malware* W32.Evol.

Como objeto de estudo, foi necessária análise do metamorfismo presente nas 63 versões metamórficas manipuladas, sendo elas divididas em 16 versões para primeira geração, 8 versões para segunda geração, 4 versões para terceira geração, 2 versões para quinta geração e 1 versão para sexta geração. Esta seção mostra que as quantidades de amostras utilizadas são suficientes para detectar as tendências comportamentais do *malware* avaliado utilizando como base os grafos de dependência.

Através da análise dos códigos das 6 gerações do Evol W32, é possível comparar cada geração entre si e perceber a mudança na estrutura de código em função das técnicas de ofuscação utilizadas. A Figura 3 mostra as 10 primeiras linhas de código do Evol W32 sem metamorfismo e a Figura 4 mostra trechos de códigos das versões metamórficas do vírus Evol W32. É possível notar que tanto o código original como as versões metamórficas possuem as três primeiras linhas idênticas. Entretanto, a partir da quarta linha ocorrem as modificações introduzidas pelo processo de metamorfismo. Como consequência, nenhuma assinatura é gerada a partir de qualquer um desses trechos de código poderia ser utilizado para identificar os outros.

```
CODE:00401000 push   ebp
CODE:00401001 mov    ebp, esp
CODE:00401003 sub    esp, 4
CODE:00401006 mov    eax, [ebp+4]
CODE:00401009 mov    [ebp+8], eax
CODE:0040100C mov    eax, [ebp+0]
CODE:0040100F mov    [ebp-4], eax
CODE:00401012 call   sub_4011C1
CODE:00401017 cmp    eax, 0
CODE:0040101A jz     loc_401142
```

Figura 3: Codigo original do Evol W32 sem metamorfismo.

```
CODE:00401000 push   ebp
CODE:00401001 mov    ebp, esp
CODE:00401003 sub    esp, 4
CODE:00401006 mov    eax, [ebp+4]
CODE:00401009 push   ecx
CODE:0040100A mov    ecx, ebp
CODE:0040100C add    ecx, 45h
CODE:0040100F mov    [ecx-3Dh], eax
CODE:00401012 pop    ecx
CODE:00401013 push   edx
```

1a Geração

```
CODE:00401000 push   ebp
CODE:00401001 mov    ebp, esp
CODE:00401003 sub    esp, 4
CODE:00401006 mov    eax, [ebp+4]
CODE:00401009 push   ecx
CODE:0040100A mov    ecx, ebp
CODE:0040100C mov    eax, [ebp+0]
CODE:0040100F mov    [ebp-4], eax
CODE:00401012 call   sub_40130E
CODE:00401017 push   ebx
CODE:00401018 mov    ebx, 0
```

2a Geração

```
CODE:00401000        push   ebp
CODE:00401001        mov    ebp, esp
CODE:00401003        sub    esp, 4
CODE:00401006        push   esi
CODE:00401007        mov    esi, ebp
CODE:00401009        push   edi
CODE:0040100A                mov    edi, 15h
CODE:0040100F        add    esi, edi
CODE:00401011        pop    edi
CODE:00401012        mov    eax, [esi-11h]
```

3a Geração

```
CODE:00401000 push   ebp
CODE:00401001 mov    ebp, esp
CODE:00401003 sub    esp, 4
CODE:00401006 mov    eax, [ebp+4]
CODE:00401009 push   ecx
CODE:0040100A mov    ecx, ebp
CODE:0040100C push   edi
CODE:0040100D mov    edi, 45h
CODE:00401012 add    ecx, edi
CODE:00401014 pop    edi
```

4a Geração

```
CODE:00401000 push   ebp
CODE:00401001 mov    ebp, esp
CODE:00401003 sub    esp, 4
CODE:00401006 push   ecx
CODE:00401007 mov    ecx, [ebp+4]
CODE:0040100A mov    eax, ecx
CODE:0040100C pop    ecx
CODE:0040100D push   ecx
CODE:0040100E mov    ecx, ebp
CODE:00401010 push   edi
```

5a Geração

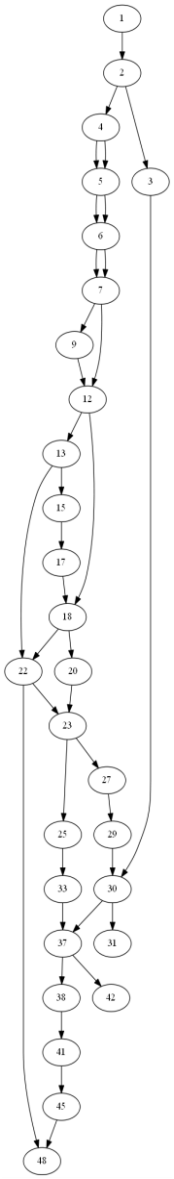
```
CODE:00401000 push   ebp
CODE:00401001 mov    ebp, esp
CODE:00401003 sub    esp, 4
CODE:00401006 push   ecx
CODE:00401007 mov    ecx, [ebp+4]
CODE:0040100A push   edi
CODE:0040100B mov    edi, ecx
CODE:0040100D mov    eax, edi
CODE:0040100F pop    edi
CODE:00401010 pop    ecx
```

6a Geração

Figura 4: Em cada geração, há a troca de instruções que são equivalentes.

Utilizando essas versões metamórficas do *malware* em estudo, foi gerado um grafo de dependência que representa um trecho de código associado ao o código fonte original do Evol W32. Este grafo é apresentado na Figura 5. Além disso, foi gerado um gráfico (Figura 6), onde é possível perceber a diferença na quantidade de arestas presentes nos grafos de dependência originais ilustrados na série 1 e nos grafos resultantes do processo de redução, representados na série 2.

Figura 5: Grafo de dependência gerado a partir do trecho de código original do Evol32.



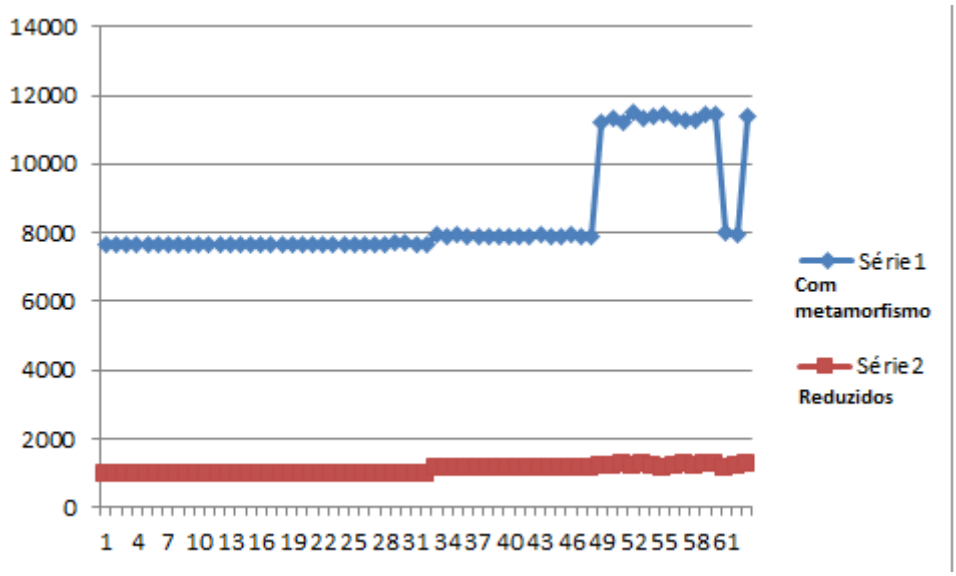


Figura 6: Gráfico comparativo entre versões com metamorfismo x reduzidos.

Pode-se ver que na série 1, apesar de uma certa linearidade inicial, a partir de um cento ponto existe um aumento expressivo na quantidade de arestas presentes no grafo de dependência gerado. Isto ocorre porque naquelas versões foram aplicadas técnicas metamórficas que substituem uma instrução simples por um conjunto de instruções que terão um resultado final equivalente à da instrução original, ou ainda pela inclusão de instruções “lixo” que aumentam o tamanho do código. A série 2 apresenta uma linearidade maior, ainda que não perfeita. O que é mais relevante é que nesta série não é obsevado o mesmo salto quantitativo observador na série 1, o que ilustra a efetividade na ação do processo de redução em eliminar elementos irrelevantes. É esta efetividade na anulação das ações das técnicas de metamorfismo de código que torna possível a utilização dos grados de dependência como base de identificação deste tipo de *malware*.

5 Conclusão e Trabalhos Futuros

Este trabalho apresentou um mecanismo para pré-processamento geração de grafos de dependência e identificação de variantes metamórficas de um *malware*. A metodologia se baseia em fazer uma identificação por grafo de dependência do *malware* e do arquivo suspeito de contaminação, buscando através da redução de grafos, detectar técnicas de ofuscação presente em códigos executáveis. Para fazer esta identificação, os programas passam por um processo no qual se obtém o código em linguagem *assembler* que seja equivalente ao código executável original, executa-se um pré-processamento dos códigos *assembler* e a geração do grafo de dependência correspondente. A partir deste ponto, é aplicada uma técnica de redução de grafos de dependência que permite remover as técnicas de ofuscação e, por consequência, reduz drasticamente a quantidade de linhas de código.

Como trabalho futuro, é possível estender a pesquisa para desenvolver um software que automatize a atividade de encontrar o máximo isomorfismo de subgrafo, visto que é a segunda parte do estudo e que detém maior atenção por ser um problema NP-difícil e que tem muitos estudos na área. Para isso, será necessário usar a metodologia em conjunto com algum tratamento. Essa busca pelo máximo isomorfismo de subgrafos permite encontrar um padrão de grafos que seja utilizado como técnica de identificação de *malware* embutidos em códigos maiores. Além disso, será possível encontrar infecções isoladas em códigos que dificilmente seriam detectadas senão usando técnicas de análise comportamental que usam grafos de dependência como base para modelagem de seus códigos.

6 Referências

Borello, J.M. Mé L.. “Code obsfucation techniques for metamorphic viruses”. Journal in Computer Virology. v.4, n. 3, p. 211-220. 2008.

Bruschi, D.; Martignoni. L; Monga M. “Code normalization for self-mutating malware. Security and Privacy”, IEEE, v. 3, n. 1, p. 46–54, 2007.

Chouchane M.; Lakhotia A. “Using engine signature to detect metamorphic malware”. In WORM ’06: Proceedings of the 4th ACM workshop on Recurring malcode, 2006.

Computer Virus. Disponível em: <http://microsoft.wikia.com/wiki/Computer_virus>. Acessado em 15 set. 2013.

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

Hex-Rays. “IDA Pro”, http://www.hex-rays.com/products/ida/index.shtml, 2012.

Hu, X. Chiueh, T.-c. e Shin, KG. “Large-scale malware indexing using function-call graphs”. Proceedings of the 16th ACM conference on Computer and communications security (CCS'09), Chicago, Illinois, USA. ACM, 2009

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

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

OllyDbg. “OllyDbg”, http://www.ollydbg.de. 2012

Notoatmodjo, G. “Detection of Self-Mutating Computer Viruses”, 2005. Disponível em: http://www.cs.auckland.ac.nz/compsci725s2c/archive/termpapers/gnotoadmojo.pdf. Acesso em: 15 set. 2013.

Rad, B. B.. Masrom, M. “Metamorphic Virus Variants Classification Using Opcode Frequency Histogram”. Latest Trends on Computers v. 1, p. 147-155. 2010.

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

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

Venkatesan, A. “Code Obfuscation And Virus Detection”, San Jose State University, May, 2008.

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

7 Cronograma

Nº	Descrição	Ago 2012	Set	Out	Nov	Dez	Jan 2013	Fev	Mar	Abr	Mai	Jun	Jul
01	Levantamento e estudo de técnicas de detecção de malware. Estudo sobre malwares metamórficos												
02	Desenvolimento de mecanismo de detecção de malware baseado em grafos de dependencia												
03	Análise dos resultados												
04	Elaboração do Resumo e Relatório final												
05	Preparação da Apresentação Final para o Congresso												
06	Reuniões períodicas com o orientador												

Atividades Realizadas

Tabela 6:Cronograma de Atividades Realizadas

1. Levantamento e estudo das abordagens baseadas no tipo de detecção de malware.
2. Levantamento e estudo das técnicas existentes para a detecção *malwares* metamórficos. Esse objetivo específico permitiu o entendimento necessário para a realização do projeto.
3. Desenvolvimento de mecanismo de identificação de comportamento malicioso baseado em grafos de dependência. O objetivo é fundamental para garantir a detecção de malware baseado em grafos de dependência.
4. Reuniões periódicas com o orientador: reuniões são fundamentais para o direcionamento, auxílio e acompanhamento do andamento do projeto.
5. Apresentação dos resultados: após pesquisas, análises e execução do projeto, serão apresentados os resultados finais obtidos com a solução proposta.