Detecção de variações de malware metamórfico por meio de normalização de código e identificação de subfluxos

Marcelo F. Cozzolino[1,2], Gilbert B. Martins[3,4], Eduardo Souto[3], Flávio E. G. Deus[2]

[1]Departamento de Polícia Federal, Setor Técnico-Científico no Amazonas
Manaus, AM – Brasil

[2] Departamento de Engenharia Elétrica, Universidade de Brasília (UNB)
Brasília, DF – Brasil

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

[4]Fundação Centro de Análise, Pesq. e Inovação Tecnológia (Fucapi)
Manaus, AM – Brasil

cozzolino.mfc@dpf.gov.br, gilbert.martins@fucapi.br,
esouto@icomp.ufam.edu.br, flavioelias@unb.br,

Abstract. *This paper presents a methodology to identify metamorphic malware. The file code is normalized and subdivided in code segments (tokens) delimited by changes in program flow. The combination of each token identifier with the two following ones generates a set of flow identifiers used to measure the similarity to a previously mapped malware code. The results show that proposed methodology is able to accurately identify the presence of metamorphic code.*

Resumo. Este artigo apresenta uma metodologia capaz de identificar *malware* metamórficos. O código de um arquivo é submetido a um processo de normalização e subdividido em trechos de códigos (*tokens*) delimitados por mudanças de fluxo do programa. A combinação do identificador de cada *token* com os dois seguintes cria um conjunto de identificadores de fluxo, que são usados para medir a similaridades com um código de *malware* previamente mapeado. Os resultados obtidos mostram que a metodologia proposta é capaz de identificar com precisão a presença de códigos metamórficos.

1. Introdução

O termo *malware* (proveniente do inglês *malicious software*) é usado para classificar um software destinado a se infiltrar em um sistema de computador alheio sem consentimento, nem conhecimento, do usuário, com o intuito de causar algum dano ou roubo de informações. Vírus de computador, *worms*, *trojan horses* (cavalos de Troia) e *spywares* são exemplos de *malware* [Skoudis 2004]. Atualmente, a técnica mais comum de detecção destas ameaças, se baseia na procura de uma sequência de bytes (assinatura) conhecida do *malware* em uma base de dados (base de assinaturas).

Como resposta a esta abordagem de detecção, alguns desenvolvedores de *malware* empregam técnicas que modificam a estrutura do código automaticamente. Tais técnicas possibilitam a alteração do código do vírus à medida que sua propagação ocorre, tornando esta nova versão do código incompatível com a assinatura associada à versão anterior [Borello e Mé 2008] [Moura e Rebiha 2009].

Malwares que empregam estas técnicas podem ser classificados em dois grupos [Notoatmodjo 2010]: Polimórficos e Metamórficos. Os polimórficos utilizam técnicas de criptografia combinadas com uma mudança contínua da chave criptográfica para garantir a diferenciação contínua dos códigos gerados. Os metamórficos não se baseiam em um processo de criptografia, mas fazem uso de técnicas de ofuscação de código mais complexas, criando novos trechos de código que mantém todas as funcionalidades originais intactas. [Rad e Masrom 2010] destacam ainda que a efetividade do metamorfismo se baseia no fato de que a maioria das assinaturas é criada com base na varredura do código, o que gera uma dependência das estruturas sintáticas apresentadas por aquele código. Assim, para evitar a detecção, novas versões do código são criadas mantendo a estrutura semântica original e incluindo alterações sintáticas que os tornam incompatíveis com a assinatura gerada originalmente.

[Bruschi et al. 2007], [Notoatmodjo 2010] e [Kim e Moon 2010] apresentam as técnicas metamórficas mais comuns de ofuscação. A partir de um código original, podem ser aplicadas modificações como: inserção de instruções e variáveis irrelevantes que não interfiram com o conjunto original; renomeação de variáveis ou troca de variáveis entre operações distintas; troca de sequências de instruções por outras equivalentes; e alterações na ordem de execução das instruções, de forma que o resultado final daquela porção de código seja mantido.

Códigos maliciosos podem ser identificados a partir da análise da sequência de instruções. Para tanto, diversas abordagens alternativas têm sido empregadas para este fim. Por exemplo, temos: abordagens baseadas na análise conjuntos de instruções que caracterizam funcionalidades maliciosas [Batista 2008]; técnicas de mineração de dados que podem ser usadas para mapear estruturas de dados comumente encontradas em *malware* [Schultz et al. 2001] ou para identificar a *engine* metamórfica empregada [Andrew et al. 2006]; técnicas de normalização de código tem sido usadas na reversão dos metamorfismos [Christodorescu 2007], na utilização de operações originalmente empregadas em otimização para "limpar" o código [Bruschi et al. 2007], ou mesmo no mapeamento de grafos de dependência [Kim e Moon 2010]. Todas estas alternativas tem como objetivo possibilitar que abordagem de assinaturas possa voltar ser aplicada com sucesso.

A abordagem proposta neste artigo se enquadra na categoria de normalização de código. Primeiramente um *malware*, apresentado em formato binário, tem seu código revertido para o formato *assembler.* Em seguida, este código é normalizado e subdividido em trechos de código (aqui denominados de *tokens*) delimitados por mudanças de fluxo do programa, dividindo o código em blocos que, mesmo reposicionados, tenham a mesma sequência de instruções executadas, preservando sua funcionalidade original.

Para cada *token* é calculado um valor identificador de seu conteúdo (denominado de *token id*) através de uma função *hash* que tem como parâmetro de entrada o seu código *assembler*. Para cada *token*, através da análise da sequência de execução do código, são determinados até dois possíveis tokens subseqüentes. Ao conjunto do *hash* deste *token* concatenado com o dos seus dois possíveis subseqüentes chamamos de "identificador de fluxo" (*flow id*). Tal identificador é usado para medir a quantidade de similaridades entre um código de *malware* previamente mapeado e um código suspeito, auxiliando no processo de identificação do *malware.*

O restante deste artigo está organizado da seguinte forma: a Seção 2 apresenta um conjunto de trabalhos relacionados, apresentando diversas tecnologias utilizadas na identificação de códigos metamórficos; a Seção 3 descreve uma visão geral da metodologia proposta neste trabalho; a Seção 4 apresenta a descrição do processo de preparação da base de referência, que armazena o conjunto de *flow id* utilizado para a identificação do *malware*; a Seção 5 apresenta o detalhamento do procedimento de detecção utilizado para analisar um arquivo sob suspeita; a Seção 6 apresenta um estudo de caso que exemplifica a utilização da metodologia e uma análise dos resultados atingidos; e a Seção 7 apresenta as conclusões deste trabalho.

2. Trabalhos Relacionados

Esta seção descreve alguns trabalhos encontrados na literatura que utilizam técnicas de normalização de código no processo de detecção de *malware* metamórficos. Todas as abordagens possuem um conjunto particular de características individuais associadas à forma como cada uma trata o problema de normalização, oferecendo uma visão mais ampla dos desafios associados a esta abordagem.

[Christodorescu et al. 2007] propõe o uso de "normalização" do código no processo de identificação de variantes de *malware* metamórfico em antivírus. Esse trabalho concentra-se em três tipos específicos de mutações metamórficas: *a)* Inserção de código irrelevante – insere instruções inúteis, mas que não interfere com o desenrolar do programa; *b)* Mudança de ordenação – muda a ordem de instruções que não alterem o funcionamento final do programa; e *c)* Compactação – a compactação deixa o código em um estado que ocupa menor espaço para só ser descompactado no momento que for ser executado, o que dificulta a identificação do código conhecido devido à impossibilidade de saber qual algoritmo de compactação foi usado. O processo de normalização descrito inicia revertendo a compactação para obter um código executável, em seguida elimina os desvios (*jumps*), que alteram a ordem das instruções, gerando um código que representa o fluxo "normal" das instruções. Por último, ocorre a eliminação das instruções irrelevantes para assim gerar um novo código que se aproxime ao máximo do código original, que então será submetido à avaliação por processos tradicionais de detecção por assinatura. As principais limitações deste trabalho estão associadas à impossibilidade de garantir que o fluxo original das instruções possa sempre ser completamente reconstruído e que restem somente as instruções realmente relevantes para o funcionamento do *malware* [Christodorescu et al. 2007].

O trabalho de [Bruschi et al. 2007] baseia-se no fato de que a maioria das transformações utilizadas por *malware*, visando disfarçar sua presença, tem como efeito colateral um aumento no tamanho código binário produzido. O código resultante pode

ser encarado como um código não otimizado, pois contará com alguns comandos irrelevantes, destinados exclusivamente a mascarar o código original. Desta forma, a normalização deste código, através de técnicas de otimização utilizadas comumente por compiladores, terá como efeito a eliminação deste código "inútil", facilitando o processo de reconhecimento. Esta abordagem se diferencia das outras por adicionar uma etapa de "interpretação de código", onde as instruções *assembler* são simplificadas ainda mais em uma linguagem com poucos operadores semânticos. O código gerado por essa etapa será tratado pelos processos de normalização e de comparação, na busca pela identificação do *malware*. Entretanto, apesar do processo de normalização, aplicado durante a execução de testes, apresentar tempos razoáveis, quando aplicado a programas de tamanho maior demandou uma grande quantidade de tempo e recursos computacionais, o que limita sua efetiva aplicabilidade [Bruschi et al. 2007].

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

A metodologia proposta neste trabalho também visa desfazer as mutações metamórficas, contidas em uma biblioteca previamente mapeada, antes de proceder à comparação do código. Entretanto, diferentemente de [Christodorescu et al. 2007], o problema de levantamento do fluxo é tratado neste trabalho possibilitando capturar de forma mais efetiva a lógica de funcionamento do código reconstruído. Além disso, o processo de normalização utilizado neste trabalho é baseado em bibliotecas de metamorfismos pré-conhecidas, apresentando ordem de complexidade inferior aos trabalhos de [Bruschi et al. 2007] e [Kim e Moon 2010].

3. Visão geral da Metodologia Proposta

O processo de detecção de um *malware* é baseado na construção de uma base de referência a partir do código normalizado do *malware* buscado, que será usada na comparação com o código também normalizado de um arquivo sob suspeita de contaminação. Na base de referência ficam armazenados os identificadores de trechos de códigos (*token id* e *flow id)* gerados após a normalização e processamento do código de

um *malware* metamórfico. Com esta base preparada, o mesmo procedimento de normalização e geração dos identificadores de trechos de códigos é executado em um arquivo suspeito de contaminação. O processo de detecção visa encontrar correspondências de sequências de trechos de códigos entre a base de referência e o conjunto de identificadores de trechos de código gerados a partir do arquivo suspeito, através da comparação de seus *flow id*. Cada correspondência de *flow id* (*flow_id matching)* é então contabilizada para indicação do nível de compatibilidade entre o código do *malware* mapeado e o código sob suspeita. O valor auferido ao término deste do procedimento de comparação é usado para determinar se o arquivo investigado está ou não contaminado. A Figura 1 ilustra uma visão geral da metodologia proposta neste trabalho e as seções seguintes detalham cada etapa.

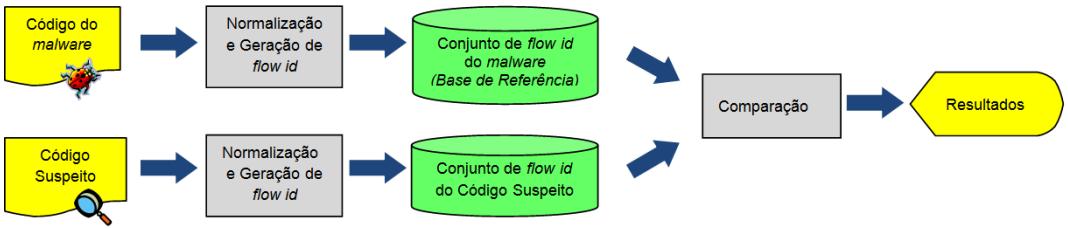

Figura 1 – Visão geral da metodologia de detecção de *malware* metamórficos

4. Preparação da Base de Referência

Como mencionado anteriormente, a base de referência é basicamente composta por identificadores de trechos de códigos normalizados (*tokens*). O processo de geração desses identificadores é realizado em cinco etapas: *disassembler*, levantamento do fluxo do programa, divisão em *tokens*, normalização e cálculo do *flow id*.

4.1. Disassembler

A base das ferramentas de análise de programas maliciosos é o processo de *disassembler*, cuja função é executar a transformação de um código binário, visando à recuperação de um código *assembler* e viabilizando a interpretação das instruções. Este processo pode ser realizado através de *disassemblers* convencionais encontrados no mercado. OllyDbg [OllyDbg 2011] e IDA Pro [Hex-Rays 2011] são exemplos de *disassemblers* compatíveis com o Windows, enquanto que *Bastard Disassembler* [SorceForge 2011] e LIDA [Schallner 2004] são exemplos de *disassemblers* compatíveis com o Linux.

4.2. Levantamento do fluxo do programa

Após o processo de *disassembler* do código, inicia-se a etapa que busca levantar a ordem de execução das instruções para construir o fluxo do programa. Para tornar isso possível, são levantadas, para cada instrução, quais as possíveis instruções subsequentes. Cada instrução terá dois endereços de saída, que serão representados pelas variáveis F1 e F2. F1 será sempre o endereço da instrução seguinte pelo posicionamento da memória (execução de código sequencial) e F2 será o endereço da instrução de desvio (em caso de desvio de código). Instruções que permitam mais de dois destinos de saída (retorno de sub-rotina ou desvios indiretos) não terão o fluxo levantado neste trabalho, devido à

complexidade desta operação em análise estática. Nestes casos, os valores de F1 e F2 serão igualados a zero, para sinalizar que o fluxo não foi levantado. Nas chamadas de sub-rotinas será considerado que uma sub-rotina terá o comportamento padrão, ou seja, retorna para a instrução seguinte. Neste levantamento não serão considerados os desvios do programa nas chamadas de sub-rotinas.

4.3. Divisão em *tokens*

A terceira etapa consiste na divisão do código em partes menores que serão usadas na fase de comparação dos códigos. Essas partes menores têm sua função semântica dentro do código preservada durante as operações de metamorfismo, passando a serem identificadas como *tokens*. A divisão do código em *tokens* deve ser feita de forma a garantir que, mesmo após o processo de metamorfismo, o código alterado permanece dentro do mesmo *token* que sofreu o metamorfismo. Metamorfismos podem além de alterar a forma de apresentação dos *tokens*, alterar a ordem dos *tokens* em que essa ordem não seja importante. Para a divisão dos *tokens* foram utilizados critérios para atender as seguintes condições:

1. Os *tokens* devem ser executados a partir da primeira linha e terminados na última linha. O objetivo é evitar que o código de um *token* comece a ser executado por uma linha que não a primeira deste. Isso faz com que em *tokens* equivalentes, códigos também equivalentes tenham sido executados;
2. A divisão dos *tokens* deve facilitar a identificação do próximo *token* a ser executado logo após a saída do *token* corrente. Caso não seja possível identificar qual o *token* seguinte de forma trivial em análise estática, este fato deve ser sinalizado.

Foram usados três tipos de delimitadores dos *tokens*: *i)* As linhas *assembler* com rótulos (*labels*); *ii)* As instruções de desvio; e *iii)* As instruções de retorno de *subrotina*. Através do fluxo de programa, levantado pela última instrução do *token*, serão identificados *tokens* seguintes, o que será usado posteriormente na etapa de comparação. A Tabela 1 mostra um exemplo de código *assembler* dividido em *tokens*, mostrando também os dois possíveis *tokens* seguintes.

Tabela 1 – Exemplo da divisão em *tokens* do código *assembler*

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

4.4. Normalização

Na etapa de normalização, os *tokens* passam por um processo que visa tornar semelhantes *tokens* derivados de um mesmo *token* inicial, desfazendo as alterações que foram implementadas pela *engine* metamórfica, de forma a se obter um código mais próximo ao "original". Entretanto, não é possível ter certeza que determinado código é derivado de um metamorfismo. Independente de ter sido originado ou não pela ação da *engine* os códigos serão revertidos. Isso não trará problemas na comparação por que esta reversão também ocorrerá no código do *malware*, antes da comparação. Além disso, existem transformações que podem ser executadas e, após vários metamorfismos sobrepostos, voltar à condição inicial, gerando uma situação "cíclica", não sendo possível definir exatamente qual seria o código inicial. Em casos assim, arbitra-se um código como sendo o inicial em todas as ocorrências, de forma a levar todas as variações metamórficas até um ponto comum.

O processo de normalização usado neste trabalho é baseado em bibliotecas de *engines* metamórficas conhecidas de *malware*. Uma *engine* metamórfica corresponde ao trecho de código do *malware* responsável pela mutação do código. A ideia é entender o processo de metamorfismo empregado pelo *malware* e desfazer uma a uma as mutações. Para garantir a volta ao código inicial, seria necessário desfazer os metamorfismos na ordem inversa em que foram feitos. Contudo, como não é possível identificar a quantidade de vezes e a ordem em que os metamorfismos foram aplicados, não existe um procedimento exato que garanta a reversão dos metamorfismos. Ciente deste problema, este trabalho usa a seguinte ordem de reversão: desfazer primeiro as mutações mais complexas (aquelas que geram mais instruções) e depois as mais simples. Essa escolha de reversão considera que um código resultante de uma mutação mais complexa tem menos chance de ser sido produzido incidentalmente, sem participação da *engine* metamórfica. Como exemplo, a Figura 2 apresenta transformações, que são partes de uma *engine* metamórfica. A reversão do metamorfismo exposto em (a), pode ser inviabilizada se a reversão do metamorfismo exposto em (b) for executado antes.

Linha	**Código com metamorfismo**	**Código Original**
1	push eax	movsb
2	mov al, [esi]	
3	add esi, 1	
4	mov [edi], al	
5	add edi, 1	
6	pop eax	
(a)Metamorfismo complexo		
Linha	**Código com metamorfismo**	**Código Original**
1	mov al, [esi]	lodsb
2	add esi, 1	
(b)Metamorfismo simples		

Figura 2 – Exemplos de metamorfismos

O código metamórfico apresentado em (b) corresponde a um subconjunto do código metamórfico apresentado em (a), linhas 2 e 3. Se estas linhas forem substituídas pelo código original em (b) – instrução lodsb, o código original em (a) – instrução movsb, não será gerado pelo processo de reversão.

Durante o processo de reversão todo trecho de código que corresponder ao resultado de um metamorfismo será revertido, não importando se o código foi realmente resultante

de um metamorfismo ou não, visto que o objetivo não é obter o código original, mas reconhecer códigos que possam ter a mesma origem independente do estado de metamorfismo.

O algoritmo varre os trechos de códigos correspondentes a um *token* visando identificar todas as instruções ou conjunto de instruções que são candidatas à reversão. Em seguida, esses conjuntos de instruções são processados de acordo com a ordem de complexidade, da maior para a menor, obedecendo à prioridade de reversão explicada anteriormente. Este processo se reinicia até que não reste nenhuma para ser revertida naquele *token*.

4.5. Cálculo de *Flow Id*

Após o processo de normalização, inicia-se a geração do conjunto de identificadores de fluxo (*flow id*). Estes elementos são gerados usando os identificadores individuais de cada *token* (*token id*) e de seus dois *tokens* subsequentes, de acordo com o levantamento do fluxo do programa (veja Seção 4.2). Para gerá-los é aplicada uma função *hash* que tem como entrada o conteúdo considerado relevante no código do *token*. Neste cálculo de *hash* são consideradas relevantes às instruções e os parâmetros que não sejam endereços de desvio ou de endereço de chamadas de sub-rotinas. Essa diferença de tratamento nos endereços visa garantir que códigos equivalentes sejam reconhecidos como tal, independente do fato do endereço de desvio dos mesmos ter mudado. Isto acontecerá quase sempre em códigos que sofreram metamorfismo, pois o código sofrerá reposicionamento devido às instruções inseridas, mudando os endereços de destino dos desvios.

5. Procedimento de Detecção

O procedimento de detecção consiste em repetir as etapas descritas na Seção 4 para cada um dos arquivos que se pretende avaliar e efetuar a comparação entre os dois conjuntos de *flow id*, como mostrado na Figura 1. O processo de comparação é detalhado nas próximas seções.

5.1. Comparação dos Códigos Normalizados

Nesta etapa ocorre a comparação do conjunto de *flow id* dos arquivos suspeitos de contaminação com os do *malware* previamente mapeado. Para cada *token* normalizado do arquivo em análise, será comparado o seu *flow id* com o *flow id* de todos os *tokens* do *malware* buscado. Cada coincidência incrementará um contador de incidência. O valor final desse contador será usado para decidir se o arquivo analisado está ou não contaminado por uma variante metamórfica.

5.2. Tratamento de Falsos Positivos

O resultado da comparação dos *flow id* pode ser positivo quando da presença do *malware* buscado ou em casos onde ocorrerem coincidências de trechos de código, conduzindo a falsas identificações do *malware*. Quanto maior for o tamanho do arquivo sob teste, maior será a possibilidade de coincidências acidentais. Podem ocorrer casos onde um programa livre de contaminação atinja uma pontuação de incidência superior a um programa menor que esteja efetivamente contaminado com o *malware*.

Para verificar a ocorrência deste tipo de *matching* indevido, deve-se executar novamente todo o procedimento de detecção sem, entretanto, processar a etapa de normalização do código suspeito. Após o término deste novo processamento, os resultados obtidos são comparados com os resultados originais para efeito de validação. Caso não exista diferença na pontuação de *matching*, isto indicará que as pontuações foram por motivo de coincidência acidentais de trechos de código uma vez que, como a etapa de normalização foi omitida, não foi necessário desfazer qualquer metamorfismo para a ocorrência de *matching*, o que prova que o código suspeito está livre de contaminação.

6. Estudo de Caso

Nos testes de validação da metodologia, foi escolhido o vírus W32 Evol [Symantec 2007]. A escolha de tal vírus foi devido ao fato deste possuir uma *engine* metamórfica que contempla as técnicas mais comuns empregadas para efeito de ofuscação de código.

Para produção de amostras de teste, a ação da *engine* metamórfica do W32 Evol foi emulada através de um programa em linguagem "C++", construído através da conversão direta do código em *assembler* desta *engine* (encontrado no código fonte do W32 Evol), para seu equivalente em linguagem "C++", que executa o processo de ofuscação de código em qualquer arquivo desejado.

O programa de metamorfismo foi executado produzindo 63 versões do *malware* "W32 Evol.a". Cada execução da *engine* metamórfica sob um arquivo será chamada de "geração". Assim, das 63 versões do *malware*, 32 versões foram feitas executando o programa de metamorfismo uma única vez (amostras de primeira geração), 16 versões foram feitas executando o programa de metamorfismo duas vezes (amostras de segunda geração), 8 versões (amostras de terceira geração), 4 versões (amostras de quarta geração), 2 versões (amostras de quinta geração) e 1 versão (amostra de sexta geração). Os experimentos apresentados nesta Seção mostram que as quantidades de amostras utilizadas são suficientes para apontar as tendências comportamentais do *malware* avaliado. O processo de avaliação é feito comparando o vírus e suas amostras metamórficas geradas com os antivírus comerciais e com a metodologia proposta neste trabalho.

6.1. Avaliação comparativa de antivírus comerciais

Para avaliar a identificação dos *malware* foram executados testes na ferramenta *VirusTotal* [Virus Total 2011] , que avalia arquivos suspeitos em 43 antivírus comerciais. Primeiramente foi testado o *malware* "W32 Evol.a" em seu estado original, que foi identificado por 39 dos 43 antivírus.

A Tabela 2 apresenta os resultados dos experimentos realizados no *VirusTotal* com 10 amostras metamórficas geradas, sendo 5 amostras de primeira geração e 5 de segunda geração. São apresentados apenas os antivírus que obtiveram uma identificação positiva, mesmo que sem especificar o *malware* corretamente.

Tabela 2 – Resultado do teste das versões metamórficas no VirusTotal

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

Os resultados dos experimentos no *VirusTotal* das variantes metamórficas foram insatisfatórios do ponto de vista de eficiência na detecção. No melhor caso, somente 5 antivírus identificaram simultaneamente os arquivos submetidos, dos quais apenas 2 antivírus (*PCTools* e Symantec) identificaram em alguma amostra como sendo o *malware* Evol W32. Esta baixa taxa de sucesso é compatível com os resultados apresentados por [Kim e Moon 2010].

6.2. Avaliação usando a metodologia proposta

Neste cenário de avaliação as 63 amostras metamórficas geradas foram comparadas com o "W32 Evol.a" usando a metodologia deste trabalho. Para cada arquivo foram feitas duas comparações: a primeira sem desfazer os metamorfismos e outra desfazendo. Exemplos dos resultados destas comparações são apresentados na Tabela 3. Além destas amostras foram também comparados 8 programas executáveis que não possuem o *malware* e três versões do *malware* W32 Evol – "W32 Evol a", "W32 Evol b" e "W32 Evol c" – disponíveis no site www.vxheavens.com. A metodologia deste trabalho foi aplicada em cada um destes arquivos.

Este trabalho optou por utilizar o valor absoluto do processo de *matching* (contador de incidência) ao contrário de fazer qualquer tipo de normalização de valores. Desta forma, a diferença na quantidade de pontos de *matching* obtida durante o processamento normal e o de validação (veja seção 5.2) será o elemento que tomaremos como base para identificação da presença ou não de contaminação. A diferença de pontuação entre os arquivos com o *malware* (veja a Tabela 3) e os sem *malware* (Tabela 4) foi tão grande que não foi necessário determinar um limiar mínimo que indique que o arquivo investigado está contaminado.

Tabela 3 – Exemplos da pontuação da comparação do "W32 Evol.a" com suas versões metamórficas, separadas por geração de metamorfismo, desfazendo (D) e não desfazendo (ND) o metamorfismo.

Número da Amostra	Geração											
	1ª (ND)	1ª (D)	2ª (ND)	2ª (D)	3ª (ND)	3ª (D)	4ª (ND)	4ª (D)	5ª (ND)	5ª (D)	6ª (ND)	6ª (D)
1	40	126	3	65	3	47	2	26	2	10	2	9
2	36	117	2	63	2	40	2	31	0	15		
3	40	130	1	60	1	36	0	18				
7	37	123	2	60	2	48						
13	42	125	3	66								
27	40	119										

No universo de amostras testadas, a menor diferença de pontuação entre o processamento normal e o de validação ocorreu na amostra de sexta geração e foi igual a 7(9-2). Foi observada uma tendência a diminuição desta diferença em cada geração do *malware* possivelmente devido a um aumento de falhas ao desfazer os metamorfismos quando em gerações de maior índice.

A Tabela 4 apresenta os resultados obtidos testando arquivos que não possuem o *malware*, para avaliação de falsos positivos. Na última coluna das Tabelas 4 está o resultado da comparação de cada arquivo com ele mesmo (auto-comparação), para validação dos resultados, certificando que a baixa pontuação na comparação não tenha sido devido à falha da metodologia em identificar os *tokens*. Nestas amostras a diferença de pontuação de matching entre desfazendo e não desfazendo os metamorfismos foi igual a zero em todos os casos.

Para validar o processo de verificação, foi realizado um experimento usando outras versões do "W32 Evol" com o código livre de qualquer tipo metamorfismo. A Tabela 5 apresenta os resultados obtidos. Podemos observar que não fez diferença desfazer ou não os metamorfismos, o que caracteriza que eles não são versões metamórficas do mesmo *malware*, mas sim versões originais (sem metamorfismo) de programas ligeiramente diferentes entre si, o que foi confirmado posteriormente por observação direta dos três códigos.

Tabela 4 – Pontuação de similaridade do "W32 Evol.a" com arquivos livres de contaminação (sem o *malware*), desfazendo (D) e não desfazendo (ND) o metamorfismo.

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

Tabela 5 – Pontuação de similaridade do "W32 Evol.a" com outras versões do "W32 Evol", desfazendo (D) e não desfazendo (ND) o metamorfismo

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

6.3. Análise dos resultados

Para melhor entendimento dos resultados obtidos durante o processo de experimentação, e apresentados nas Tabelas 3 e 4, destacam-se as seguintes observações:

1) Cada ponto na comparação pode ocorrer por duas razões: pela presença do *malware* ou, por coincidências acidentais dos *flow id*. Coincidências podem ser provenientes do arquivo analisado usar estruturas semelhantes as existentes no *malware* ou por colisão no algoritmo de *hash* do cálculo do *flow id*;

2) Um *malware* comumente estará em uma de três situações:

   a) Acrescido ao código do programa hospedeiro tornando-o maior. Neste caso, a pontuação do arquivo avaliado será provavelmente igual à soma da pontuação do *malware* isolado mais a pontuação acidental do programa hospedeiro original;

   b) Substituindo uma parte inerte (sem código ou dados) do programa original, para dificultar a detecção do *malware* pelo aumento do tamanho do executável original. Neste caso, o *malware* deve estar sobrepondo uma região onde não há código e, portanto, é muito improvável que haja uma coincidência acidental (que estaria sendo destruído pelo *malware*). Assim a provável pontuação de *matching* do arquivo analisado será também a soma da pontuação do *malware* isolado mais a pontuação acidental do arquivo hospedeiro original (sem o *malware*);

   c) Não possuir programa hospedeiro, estando situado em arquivo só com o *malware*;

3) Não foram feitos testes nas duas primeiras condições (item 2, "a" e "b"), com arquivo hospedeiro infectado, tendo sido feito unicamente com o *malware* isolado, ficando o teste nas outras condições como sugestão para trabalhos futuros;

4) Para avaliação de ocorrências de falsos positivos, a metodologia foi testada em 8 arquivos sem o *malware* (livre de contaminação), todos de tamanho igual ou inferior a 251KB. Neste universo ocorreu em apenas um caso com 1 (um) *matching* por coincidência acidental;

Tabela 6 – Diferenças máxima e mínima entre a pontuação desfazendo (D) e não desfazendo (ND) o metamorfismo e pontuação máxima e mínima para cada geração do arquivos de *malware*.

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

5) Dentre os arquivos com *malware* testados foram levantadas as pontuações máxima e mínima conseguidas dentre as amostras de cada geração desfazendo o metamorfismo, para cada geração do *malware*. Tal critério pode ser usado para a identificação do *malware*, conforme mostra a Tabela 6

6) Como complemento ao critério de identificação do *malware* também foram empregadas as diferenças máxima e mínima desfazendo e não desfazendo o metamorfismo, para cada geração do *malware* (ver Tabela 6).

7) Se as amostras apresentadas na Tabela 3 não estivessem sozinhas, e sim anexadas em algum arquivo hospedeiro (casos 2.a e 2.b), a diferença de pontuação permaneceria a mesma. Isto ocorreria porque as pontuações extras decorrente de identificação de estruturas similares (acidentais) ao vírus estariam presentes nos dois momentos do arquivo avaliado (normalizado e não normalizado). Isto seria verdade com os 8 arquivos livre de contaminação com os quais foram feitos os testes (veja tabela 4), mas nem sempre esta diferença será nula, sendo sugerida a avaliação deste comportamento para arquivos maiores em trabalhos futuros.

8) Nos casos testados, não houve interseção entre as regiões de máximo e mínimo nem de pontuação e nem de diferença entre as diferentes gerações, sugerindo ser possível identificar inclusive a geração do *malware* presente no arquivo pela pontuação final, dependendo do intervalo. No entanto, principalmente nas gerações mais elevadas, as amostradas utilizadas foram pequenas para tal afirmação, necessitando de uma avaliação mais ampla para uma conclusão categórica.

9) Analisando os resultados obtidos com os testes em arquivos livres de contaminação, de até 251KB de tamanho, os resultados sugerem que um valor de pontuação entre 1 e 9 ou uma diferença de pontuação (processo de validação, conforme seção 5.2) entre 0 e 7, poderiam ser usados como limiar para identificação deste *malware*. Isto foi verdade para o caso particular deste *malware* com os arquivos usados nos testes. Para determinar um algoritmo que determine os valores deste limiar para um caso genérico serão necessários muito mais testes com amostras muito mais variadas.

7. Conclusões

Os resultados obtidos nos experimentos foram bastante promissores, tendo identificado 100% dos arquivos que continham o *malware* e 0% de falsos positivos com os arquivos testados sem *malware*. Durante os experimentos foram identificas que as três versões do *malware* testado não eram versões metamórficas de um mesmo *malware*, mas sim versões ligeiramente diferentes do *malware*, fato confirmado posteriormente através da observação direta dos três códigos, reforçando os indícios de validade da metodologia.

Os resultados obtidos indicam o potencial da metodologia proposta e justificam a continuidade de novas pesquisas. Questões ligadas à cifragem e compactação de código não são tratadas nesta metodologia. Para que a metodologia seja aplicável nestes casos, será necessário usar a metodologia em conjunto com algum outro tratamento específico. Quando não for possível acessar o código decifrado em análise estática essa metodologia poderia ser aplicada de forma dinâmica como, por exemplo, usando o conteúdo da memória. Outra evolução interessante da metodologia é analisar o seu funcionamento quando aplicada a arquivos maiores e sem contaminação, visto que estes podem gerar pontuações de *matching* altas por simples coincidência. Caso ocorram situações de falso positivo, buscar soluções para minimizá-las. Um algoritmo de normalização da pontuação, relevando o tamanho ou número de *tokens*, do arquivo analisado e do *malware* buscado, pode se fazer necessário. A questão da ordem em que são desfeitas as alterações metamórficas pode ser aprimorada inclusive prevendo a possibilidade de testar em várias ordens diferentes, gerando vários *flow id* diferentes para um mesmo *token*.

8. Agradecimentos

Este trabalho foi parcialmente desenvolvido com o apoio do ao Programa Nacional de Segurança Pública com Cidadania (PRONASCI) e da Fundação de Amparo à Pesquisa do Estado do Amazonas (FAPEAM).

Referências

Andrew, W., Mathur, R., Chouchane, M. R. e Lakhotia, A. (2006), "Normalizing Metamorphic Malware Using Term Rewriting", Center for Advanced Computer Studies, University of Louisiana at Lafayette.

Batista, E. M. (2008) "ASAT: uma ferramenta para detecção de novos vírus", Universidade Federal de Pernambuco.

Borello, J. e Mé, L. (2008) "Code obfuscation techniques for metamorphic viruses", Journal in Computer Virology, volume 4, número 3.

Bruschi, D. Martignoni, L. Monga, M. (2007) "Code Normalization for Self-Mutating Malware" , IEEE Security & Privacy, volume 5, número 2.

Christodorescu, M. Jha, S. Kinder, J. Katzenbeisser, S. Veith, H. (2007) "Software Transformations to Improve Malware Detection", Journal in Computer Virology, número 3, páginas 253 – 265.

Hex-Rays. (2011) "IDA Pro", http://www.hex-rays.com/products/ida/index.shtml.

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.

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

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

OllyDbg. (2011) "OllyDbg", http://www.ollydbg.de.

Rad, B. B. e Masrom, M. (2010) "Metamorphic Virus Variants Classification Using Opcode Frequency Histogram", Latest Trends on Computers, volume 1.

Schultz, M. G. Eleazar, E. Erez, Z. Salvatore, J. S. (2001) "Data mining methods for detection of new malicious executables", Proceedings of the 2001 IEEE Symposium on Security and Privacy, páginas 38–49.

Skoudis, E. (2004) "Malware: Fighting Malicious Code", Prentice-Hall, 2004

SorceForge. (2011) "Bastard", http://sourceforge.net/projects/bastard/.

Symantec. (2007) "W32.Evol", http://www.symantec.com/security_response/writeup.jsp?docid=2000-122010-0045-99.

Schallner , M. (2004) "LIDA", http://lida.sourceforge.net/.

Virus Total. (2011) "Virus Total", http://www.virustotal.com/en/virustotalf.html.

