# Detecção de *malware* metamórfico baseada na indexação de grafos de dependência de dados

**Luis Rojas Aguilera**[1], **Eduardo Souto**[1], **Gilbert Breves Martins**[2]

[1]Instituto de Computação (ICOMP) – Universidade Federal do Amazonas (UFAM)
Av. Rodrigo Otávio, 6200 - Japiim, Manaus, AM, Brasil

[2]Campus Manaus Distrito Industrial (CMDI) – Instituto Federal do Amazonas (IFAM)
Av. Danilo de Matos Areosa, 1672 - Distrito Industrial, Manaus, AM, Brasil

{rojas,esouto}@icomp.ufam.edu.br,gilbert.martins@ifam.edu.br

***Abstract.*** *Metamorphism and code mutation have been used successfully by malware writers to generate obfuscated codes without altering the original features, making them more difficult to detect. This work presents an approach for identifying metamorphic malware through extraction of characteristics on Data Dependency Graphs, to construct a classification index that is able to quickly and accurately recognize if a certain suspicious code belongs to a family of malware. Experimental results on 3045 metamorphic virus samples show higher average accuracy rates than most commercial antiviruses.*

***Resumo.*** *O metamorfismo e a mutação de código têm sido utilizados com sucesso pelos criadores de malware para gerar códigos obfuscados sem alterar as funcionalidades originais, tornando-os mais difíceis de detectar. Este trabalho apresenta uma abordagem para a identificação de malware metamórfico através extração de características a partir de Grafos de Dependência de Dados, para a construção de um índice de classificação que seja capaz de reconhecer de forma rápida e precisa se um determinado código suspeito pertence à uma família de malware. Os resultados experimentais sobre 3045 amostras de vírus metamórficos apresentam taxas médias de acurácia superiores a maioria dos antivírus comerciais.*

## Introdução

Apesar dos crescentes investimentos e esforços realizados pelas companhias de segurança na criação de soluções de detecção de software maliciosos (e.g. antivirus, antispyware e adware), a infecção de *malware* continua sendo uma das principais ameaças a segurança dos sistemas computacionais no mundo. A sofisticação dos ataques e o aumento cada vez maior das famílias de *malware* tem tornado a defesa contra cibercriminosos uma tarefa ainda mais difícil. De acordo com a empresa AV-Test [AV-Test 2015], considerando somente os anos de 2013 e 2014, a quantidade de novas amostras de *malware* aumentou de 83 para 142 milhões, representando uma taxa de crescimento superior à 71%. Um relatório produzido pela Symantec descreve um aumento de 274 milhões de amostras de vírus em 2014 para 357 milhões em 2016 [Symantec 2017]. Esses estudos reconhecem a falta de capacidade das ferramentas de detecção de *malware* existentes em lidar com as técnicas usadas pelos atacantes para evadir tais ferramentas.

 © 2017 Sociedade Brasileira de Computação

Essa dificuldade ocorre principalmente porque os criadores de programas maliciosos empregam várias técnicas de evasão como polimorfismo e metamorfismo de código para criar novas variantes de um *malware* a partir de uma mesma instância de código original, gerando diversas estruturas e padrões de código aleatórios [Martins et al. 2016]. Tal nível de variedade pode ser obtido de forma automatizada, a uma taxa exponencial, o que torna a criação de modelos de identificação de *malware* um processo ainda mais difícil.

Para lidar com os desafios colocados pelos *malware* metamórficos, novas abordagens para analisar a semântica e o comportamento de programas suspeitos tem sido propostas. Algumas dessas abordagens incluem: $a)$ modelagem estatística dos padrões de código gerados pelos motores metamórficos [Lin and Stamp 2010] [Singh et al. 2015] [Kuriakose and Vinod 2014]; $b)$ análise das distribuições das ocorrências de sequências de *opcodes* de instruções [Rad et al. 2012]; $c)$ análise estatística composta pela combinação de métodos de ranqueamento de características de grupos de *opcodes* de instruções [Kuriakose and Vinod 2015]; e $d)$ análise de representações intermediárias que expressam as semânticas do código, tais como: grafos de controle de fluxo [Hu et al. 2009] [Alam et al. 2015a], grafos de chamadas a APIs do sistema [Ahmadi et al. 2013] e grafos de dependência de dados [Kim and Moon 2010] [Martins et al. 2016].

Dentre as abordagens citadas, o uso de Grafos de Dependência de Dados (GDDs) se apresenta como uma alternativa promissora para detecção de *malware* metamórficos, visto que as relações semânticas representadas pelos GDDs se mantêm praticamente inalteradas mesmo quando uma nova versão de *malware* é criada. Um GDD é uma representação que usa uma notação baseada em grafos para descrever todas as relações de dependências de dados existentes entre as instruções de um programa de computador. Liu et al. [2006] e Kim e Moon [2010], por exemplo, demonstraram que é possível usar GDDs para identificar efetivamente pelo menos cinco tipos diferentes de técnicas de ofuscação de código: *i)* alteração de formato; *ii)* mudança no nome das variáveis; *iii)* reordenamento de estruturas; *iv)* troca de estruturas de controle; e *v)* inserção de instruções inócuas (ou código lixo).

Entretanto, o uso de grafos para identificação de códigos maliciosos requer um processo de comparação capaz de diferenciar grafos gerados a partir de programas benignos, daqueles gerados a partir de *malware* previamente identificados. Devido ao processo de metamorfismo de código, o casamento (*matching*) de grafos deve ser executado com foco na identificação do nível de similaridade entre subgrafos (máximo isomorfismo de subgrafo), que é reconhecidamente um processo de alta complexidade computacional [Garey and Johnson 1990].

Neste contexto, este trabalho propõe uma abordagem para a detecção de códigos maliciosos metamórficos baseado na indexação de grafos de dependência de dados de um programa. Cada *malware* é representado por um conjunto de grafos de dependência de dados. Os nós dos grafos são rotulados com base na semântica das instruções obtidas do código assembly do *malware*. Modelos de classificação são gerados para simplificar o processo de comparação de grafos, usando somente as características estruturais dos grafos. Tais modelos são utilizados como estruturas na forma de índices que servem para detectar novas instâncias baseado na comparação com os padrões presentes em instâncias conhecidas. Os resultados experimentais sobre 3045 amostras de vírus metamórficos

 © 2017 Sociedade Brasileira de Computação

apresentam taxas médias de acurácia superiores a maioria dos antivírus comerciais.

O restante deste documento está organizado como segue: A Seção 2 descreve alguns trabalhos que empregam abordagens baseadas em grafos para lidar com o problema da identificação de *malware* metamórfico. A Seção 3 introduz os aspectos técnicos envolvidos no reconhecimento de códigos maliciosos ofuscados e conceitos básicos à respeito de grafos de dependência de dados. A Seção 4 descreve alguns resultados experimentais incluindo a comparação da abordagem proposta com soluções comerciais. Por fim, a Seção 5 apresenta as considerações finais e propostas de trabalhos futuro.

## Trabalhos Relacionados

Em função de sua natureza não trivial, diversas abordagens foram propostas para tratar o problema de identificação de *malware* metamórficos. Muitas dessas abordagens são baseadas no uso de modelos de detecção baseados em grafos.

Xin Hu et al. [Hu et al. 2009] propõe um sistema de gerenciamento de bases de dados de *malware* denominado de SMIT (*Symantec Malware Indexing Tree*), cuja detecção é baseada na comparação de grafos de chamadas de funções de *malwares* conhecidos. Cada instância de *malware* é representada por um grafo, que é consultado no momento da tentativa de identificação, através de uma busca baseada na regra do vizinho mais próximo (*K-Nearest Neighbor - KNN*) [Paredes and Chávez 2005]. Para diminuir o tempo de execução das consultas, Xin Hu et al. empregam um método que calcula a similaridade dos grafos usando características estruturais à nível de instruções e um mecanhismo de indexação de resolução múltipla baseado na utilização de vetores de características adaptáveis.

Kim e Moon [Kim and Moon 2010] propõem a utilização de grafos de dependências de dados, como base de comparação, pois as relações de dependência costumam ser resistentes ao processo de metamorfismo. Para determinar se uma instância é uma variante metamórfica, o processo de comparação é tratado como a solução do isomorfismo máximo de subgrafos [Raymond and Willett 2002]. Heurísticas baseadas na utilização de algoritmos genéticos são utilizadas para diminuir o tempo de comparação. Apesar de efetiva, esta abordagem deixa de explorar outras características relevantes que poderiam enriquecer o modelo de detecção. [Martins et al. 2016] explorou esta limitação, introduzindo a classificação de nós baseado na semântica e função das instruções correspondentes, dividindo estes nós em três grupos: $a$)carga, para nós que iniciam uma cadeia de dependência, $b$)processamento, onde os conteúdos das variáveis são transformados e $c$)decisão, onde o fluxo do programa muda baseado em uma condição. Os resultados desse trabalho mostram uma diminuição tanto no tamanho dos grafos manipulados, como na variância dos resultados obtidos no processo de comparação.

Eskandari and Hashemi [Eskandari and Hashemi 2012] estudaram os padrões semânticos em códigos executáveis de instâncias metamórficas e propõem a utilização de grafos de controle de fluxo que modelem as sequências de chamadas de APIs do sistema. Com base nestes grafos é construído um vetor de características composto de pares ordenados de arestas e nós contendo as chamadas. Tais vetores são utilizados no treinamento e teste de diferentes classificadores como árvores de decisão, floresta de árvores, *Naive Bayes*, entre outros. Resultados experimentais monstram um melhor desempenho dos modelos criados utilizando florestas de árvores.

Alam et.al [Alam et al. 2015b] apresentam uma estrutura chamada de Grafo de Controle de Fluxo Anotado (GCFA), construído a partir das funções do código binário de instâncias de *malware*. Operações comumente utilizadas em código assembly são mapeadas a classes que são utilizadas para anotar os GCFAs. Uma instância é rotulada como *malware* se os padrões coincidem com os presentes numa base de referencia de GCFAs de *malware*s conhecidos.Resultados experimentais apresentam uma taxa de detecção do 98.9% e 4.5% de falsos positivos.

Apesar de todos os trabalhos apresentados nesta seção usarem grafos, cada uma das abordagens propostas usam grafos para modelar características distintas (chamada de funções, relações de dependência entre instruções, entre outras), algumas com maior ou menor carga semântica (i.e representatividade do funcionamento e finalidade do codigo original). Além disso, as estratégias de recuperação da informação e comparação variam desde o uso de heurísticas até esquemas de indexação e modelos de classificação para diminuir o tempo de processamento, dada a complexidade inerente ao processo de comparação de grafos.

Por meio da combinação das características mais interessantes dessas abordagens, este trabalho apresenta uma metodologia de identificação de *malware* metamórfico através extração de características baseadas em Grafos de Dependência de Dados, para a construção de um índice de classificação que seja capaz de reconhecer de forma rápida e precisa se um determinado código suspeito pertence à uma família de *malware*.

## Detecção de Malware Metamórfico usando GDD

Antes de apresentar a abordagem proposta, esta seção inicia definindo os principais conceitos utilizados em sua construção.

Um Grafo de Fluxo de Controle (GFC) é uma representação que usa notação de grafo para descrever todos os caminhos que podem ser executados por um programa de computador. Em tal grafo, cada nó representa um bloco básico de instruções, isto é, uma região de código sequencial sem qualquer salto de execução. Dessa forma, o destino de um salto denota o começo de um bloco, ou seja, qualquer salto termina em um bloco. Arestas direcionadas são usadas para representar tais saltos na estrutura de controle. Ainda há dois blocos especiais, o bloco de entrada e o bloco de saída, de onde se começa e termina o fluxo, respectivamente. Formalmente um grafo de controle de fluxo pode ser definido como [Alam et al. 2015b]:

**Definição 1** *Um grafo dirigido $G = (V, E)$, onde $V = V(G)$ é o conjunto de vértices do grafo que representa os blocos básicos em um programa e $E = E(G)$ é o conjunto de arestas do grafo que representa as funções de chamadas entre blocos básicos de um programa.*

Um Grafo de Dependência de Dados (GDD) é um grafo dirigido que representa as dependências entre partes de um código fonte (instruções) [Ferrante et al. 1987] baseado na utilização dos espaços de memória e o fluxo dos dados. Neste trabalho, os grafos de dependência são construidos a partir das relações existentes entre as instruções no código assembly de um binário, onde cada instrução é representada por meio de um vértice e as relações são expressadas por meio de arestas direcionadas denominadas de arestas de dependência. Para que o sentido da aresta indique de quais outros vértices, um vértice

 © 2017 Sociedade Brasileira de Computação

em particular depende, este trabalho adota uma definição para arestas de dependência adaptada daquela apresentada em Kim e Moon [Kim and Moon 2010]:

**Definição 2** *Sejam $v_i \in V$ e $v_j \in V$, onde $G = (V, E)$ é um GDD. Caso exista ao menos uma variável $x$ tal que $x$ é usada em $v_j$ e o seu valor é estabelecido em $v_i$, então existe uma aresta de dependência $e_x \in E$ saindo de $v_j$ e chegando em $v_i$.*

### Visão Geral da Abordagem Proposta

A abordagem proposta para detecção de *malware* metamórfico é baseada na comparação de GDDs de programas binários. Para construir os GDDs a partir de um código binário é necessário submetê-los a um processo de engenharia reversa. Neste processo, o código é transformado de linguagem de máquina para a linguagem assembly (*disassembler*), transcrevendo as instruções enviadas ao processador para os seus mnemônicos em assembly (*asm*). Em seguida, o código assembly é usado para identificar todas as funções do programa. Por razões de desempenho, a abordagem cria um GFC para cada função e um GDD para cada GFC criado.

Para simplificar o processo de *matching* de grafos, um vetor baseado nas características estruturais de cada GDD é construido. Tais vetores são usados na construção de modelos que identificam os padrões presentes em *malware*s conhecidos. Na etapa de análise e identificação de instâncias suspeitas, os passos anteriormente descritos até a extração de caracteristicas do GDD são executados e os vetores de características resultantes são submetidos aos modelos treinados. A Figura 1 exibe um diagrama correspondente ao passo a passo da abordagem proposta. Explicações mais detalhadas de cada um dos componentes são apresentadas nas seções seguintes.

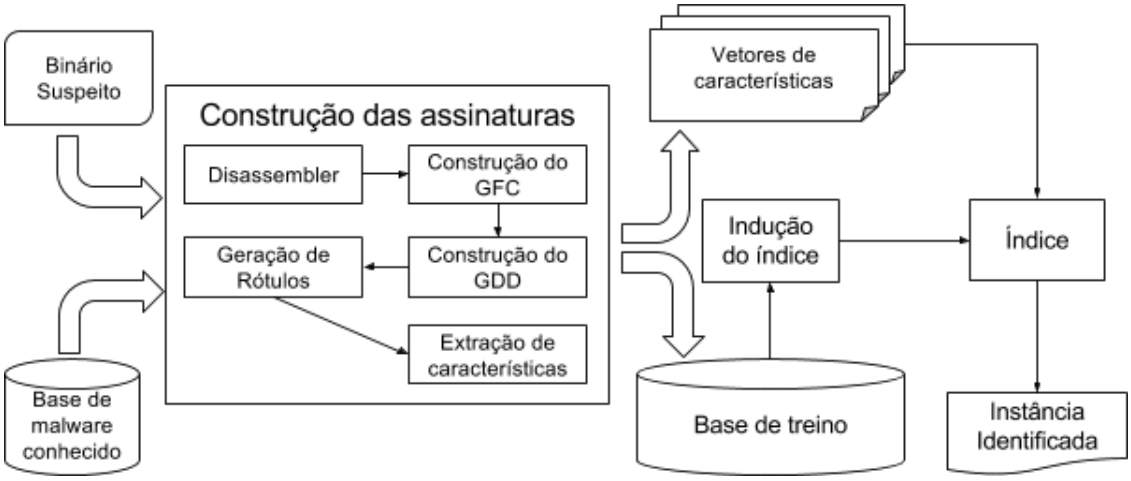


**Figura 1. Visão geral da abordagem proposta para detecção de *malware* metamórficos.**

### Construção dos Grafos de Dependência de Dados

Para obter o GDD a partir de um código executável, a abordagem proposta emprega um processo de engenharia reversa para a reconstrução de um código assembler equivalente. Neste trabalho, este processo é feito através do Radare2 [Radare2 2017], uma ferramenta para análise de código binário que, além de efetuar o disassembler, gera os grafos de

chamadas de funções e de controle. Com o objetivo de obter uma melhor diferenciação entre as amostras, a saída deste processo gera um GFC para cada função no código. Esta análise é efetuada apenas no *payload* do código binário, excluindo as seções de dados e código externo como DLLs e funções importadas.

A próxima etapa utiliza o GFC para identificar as cadeias de dependência de dados. Isto é feito pela análise do fluxo de dados de forma a identificar todas as variáveis declaradas e as partes do código onde estas são manipuladas. Nesta fase, o GFC obtido previamente é percorrido ao longo das arestas seguindo um algoritmo iterativo de análise de dados baseado na abordagem conhecida como $worklist$ [Cooper et al. 2004].

O algoritmo $worklist$ proposto, exibido em Algoritmo 1, mapeia as cadeias de dependência com base em duas funções: *In()* e *Out()*. A função *In()* contém o mapeamento de todas as variáveis usadas em todos os blocos que possuem uma aresta partindo deles e chegando no bloco atual (bloco em análise). A função *Out()* mantém o registro das modificações introduzidas no bloco atual para controlar novos mapeamentos. Assim, a origem da cadeia de dependência é sempre atualizada caso exista uma instrução no bloco atual que modifique uma variável, atualizando os resultados que serão usados na análise do próximo bloco. Como consequência disso, blocos isolados não serão considerados na construção do GDD.

No início do processamento de um bloco, a primeira ação a ser tomada é a cópia do estado atual da função *In()* para a função *Out()*. A partir deste momento, para cada instrução no bloco em análise, uma das seguintes ações pode ser tomada:

i. Caso a variável esteja sendo manipulada pela primeira vez, um vértice correspondente a essa instrução é inserido no GDD e uma nova entrada na função *Out()* é inserida para futuras manipulações daquela mesma variável;

ii. Se a variável manipulada já estiver inserida *Out()* e o seu conteúdo não é alterado pela instrução atual, uma das seguintes ações são tomadas: a) caso a instrução de origem já possua um nó inserido no GDD é criado um novo nó correspondente a instrução atual; b) caso contrário, são criados dois nós correspondentes as instruções origem e atual. Em ambos os casos, uma nova aresta é criada ligando o nó origem e nó correspodente a instrução atual.

iii. Quando a instrução estiver alterando o conteúdo da variável, são executadas as mesmas operações descritas no item *ii*, e adicionalmente a instrução atual substituirá a instrução de origem na função *Out()*, visto que esta instrução passará a ser a nova origem da cadeia de dependência desta variável.

As Figuras 2 e 3 exibem um exemplo de um GFC e seu GDD correspondente. No GDD apresentado na Figura 3 existem arestas partindo dos nós referentes as instruções $3$ e $1$ que chegam no nó referente a instrução $4$, pois a instrução $4$ utiliza o registrador $eax$ que foi manipulado anteriormente nas instruções $3$ e $1$.

### Construção de Assinaturas

Nos modelos tradicionais de detecção a assinatura de um *malware* é definida como uma cadeia de bits única que representa cada amostra. Neste trabalho, os vetores de características ($vc$) extraídos a partir dos GDDs são considerados as assinaturas das amostras sementes, isto é, amostras de código de *malware* a partir dos quais as variantes me-

 © 2017 Sociedade Brasileira de Computação

**Algoritmo 1** Abordagem Worklist para construção do GDD

```
function BUILDGDD (GFC)
    Out(s) ← ∅ for s in GFC
    V ← ∅
    E ← ∅
    W ← ∅
     W.Push(GFC.entryBB)
    while W ≠ ∅ do
        s ← W.Pop()
        In(s) ← ∩_{s'∈preds(s)} Out(s')
        temp ← In(s)
        for Instruction i in s do
            for Variable v read by i do
                if v exists in temp then
                    if i not in V then V.Add(i)
                    end if
                    if temp[v] not in V then V.Add(temp[v])
                    end if
                    E.Add(Vertex(i, temp[v]))
                end if
            end for
            for Variable v modified by i do
                temp[v] ← i
                if i not in V then V.Add(i)
                end if
            end for
        end for
        if Out(s) ≠ temp then W.Push(succs(s))
        end if
    end while
     GDD = Graph(V, E)
     return GDD
end function
```

tamórficas podem ser criadas.

Uma vez que o GDD é construído, o próximo passo consiste em atribuir um rótulo a cada nó do GDD. Este processo tem por objetivo reduzir os efeitos da técnica de obfuscação: substituição simples de instruções.

A construção dos rotulos é baseada na classificação das partes da instrução (i.e mnemonic e operandos) seguindo o padrão de nomenclatura $r_w = R_m_R_o$ para um rótulo $r_w$ qualquer, onde $R_m$ é a classe do mnemônico e $R_o$ as classes dos operandos separadas pelo símbolo de sublinhado.

A classificação dos mnemônicos segue o padrão adotado pela arquitetura de referência x86 [Lejska 2017]. Existem um total de 60 classes para 709 mnemônicos. A Tabela 1 apresenta algumas das classes de mnemônicos utilizadas.

A classificação dos operandos segue o padrão adotado pela infraestrutura de compilador LLVM (*Low Level Virtual Machine*) [Lattner and Adve 2004], também adotada pelo disassembler Capstone [Nguyen Anh Quynh 2014] [Capstone-Disassembler 2017].

Um exemplo da atribuição dos rótulos a cada nó do GDD pode ser observado

**Tabela 1. Exemplos de classificações dos mnemônicos propostas na referência x86asm[dot]net.**

| Classes | Mnemônicos |
|---|---|
| branch | jnl, jnae, call, jnp, ret, loopne, jno, jb, jnb, js, alter, jpe, ja, jmpe, retn |
| stack | push, popa, pushad, pushf, pushal, pop |
| datamov | setne, movd, setge, movlps, movbe, cmovo, sete, fistp |
| control | fdisi, wait, fstcw, hint, nop,fneni nop, fwait, ud2 |

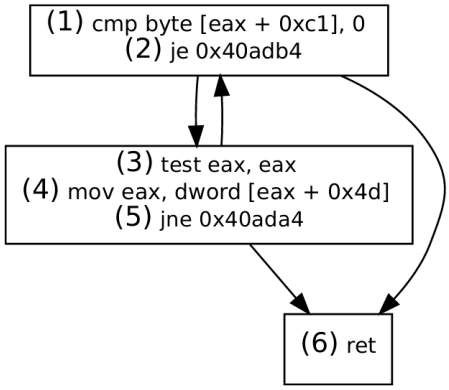


**Figura 2. Exemplo de um GFC**

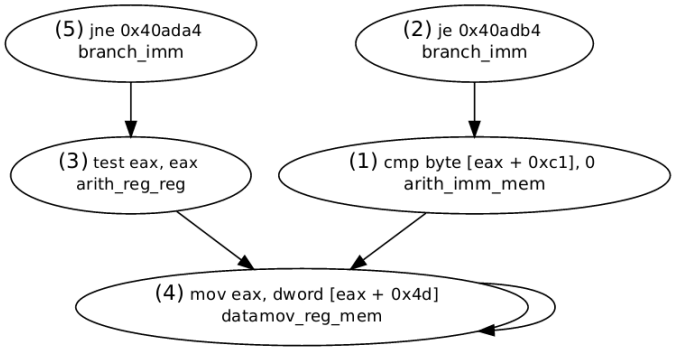


**Figura 3. GDD rotulado extraído do GFC na Figura 2**

na Figura 3. Na instrução $(3)$, o mnemonic $test$ recebe a classe $arith$ e os operandos $eax, eax$ recebem a classe $reg$.

Na medida em que o GDD é rotulado o seu vetor de características é atualizado. A abordagem proposta utiliza o padrão de nomenclatura $c_w = F_e_F_r$ para uma característica $c_w$ qualquer extraída do GDD, onde $F_r$ representa um rótulo qualquer e $F_e$ corresponde aos prefixos derivados da estrutura do grafo $n$, $i$ e $o$, sendo $n$ a quantidade de ocorrências de $F_r$ no grafo, $i$ a quantidade de arestas entrando em nós contendo $F_r$ e $o$ a quantidade de arestas saindo de nós contendo $F_r$.

A Tabela 2 mostra um exemplo de um vetor extraído do GDD presente na Figura 3. Como o grafo contém dois nós com o rótulo *branch_imm*, então *n_branch_imm* possui o valor 2. Por outro lado, *i_arith_reg_reg* é atribuído o valor 1, pois existe apenas uma aresta entrando em nós que contém o rótulo *arith_reg_reg*.

## Construção dos Índices

O método para detecção de *malware* proposto neste trabalho é baseado na comparação de vetores de características extraídas a partir dos GDDs das funções presentes em códigos binários suspeitos com aqueles de instâncias de *malware* conhecidos.

Devido aos GDDs serem menos suscetíveis à ação do metamorfismo de código, é esperado que instâncias geradas a partir de uma mesma semente sejam representadas

**Tabela 2. Exemplo do vetor de características obtido do GDD na Figura 3.**

| n_branch_imm | i_branch_imm | o_branch_imm | n_arith_reg_reg | i_arith_reg_reg | ... |
|---|---|---|---|---|---|
| 2 | 0 | 2 | 1 | 1 | ... |

 © 2017 Sociedade Brasileira de Computação

por um mesmo GDD e o seu vetor de características correspondente. No entanto, o processo de construção dos GDDs apresenta deficiências inerentes ao processo de engenharia reversa. Por este motivo, o processo de comparação de grafo é baseado em modelos baseados em padrões estatísticos ao invés de comparações exatas. Os modelos estatístico de identificação são gerados utilizando algoritmos de aprendizagem de máquina baseados em árvore de decisão.

Assim, o problema de identificar uma instância metamórfica gerada a partir de uma semente comum é visto como um problema de classificação, no qual as classes são as instâncias sementes de códigos maliciosos. Desta forma, dado um vetor de características de um GDD de um binário suspeito, o modelo deve ser capaz de reconhecer a classe (família de *malware*) a qual este binário pertence.

A escolha dos algoritmos de árvores de decisão se justifica por estes apresentarem um equílibro entre variância e viés [Munson and Caruana 2009]. Como os GDDs apresentam uma baixa variabilidade, é conveniente adotar um modelo que generalize pouco sem apresentar *overfitting* [1], sem perder as relações de relevância entre as características e as saídas esperadas.

Para a construção dos modelos mais precisos, o vetor de característica é submetido a um processo de redução com o objetivo de eliminar características com baixa variância. Uma vez selecionadas as características, um processo de treinamento supervisionado de modelos de classificação tradicional é executado [Kotsiantis et al. 2007]. A identificação da classe é definida pelo $hash\ md5$ da instância do *malware* a partir do qual foi extraído o vetor de característica.

Para identificar se uma instância suspeita pertence a alguma família de *malware* todos os passos descritos para a construção dos vetores de características são aplicados no código binário da instância em análise. Finalmente, os vetores obtidos são submetidos ao modelo treinado para obter a classe à qual pertence a instância de entrada.

## Experimentação e Resultados

Com o objetivo de avaliar a abordagem proposta um conjunto de experimentos foram realizados. Devido ao dataset de vetores de características utilizado apresentar um tamanho superior aos 8GB quando carregado em memoria e para aproveitar a capacidade de procesamento em paralelo fornecida pelo framework utilizado para treino, estes experimentos foram executados em um computador com 128 GB de RAM e 62 núcleos físicos. Os algoritmos utilizados foram implementados na linguagem de programação Python versão 3.5.

### Geração de instâncias metamórficas

A base de dados utilizada para a construção dos índices é composta pelos vetores de características extraídos dos GDDs das funções presentes no código *assembly* de instâncias de *malware* obtidos no repositório público de *malware Malshare* [Malshare 2017], os quais foram usados como sementes para gerar instâncias metamórficas.

Para gerar as instâncias metamórficas foram utilizados as ferramentas *Revert4* e *Code Pervertor* disponíveis em *VX Heaven* [VXHeaven 2017]. Estas ferramentas apli-

[1] quando o modelo estatístico se ajusta em demasiado ao conjunto de dados/amostra.

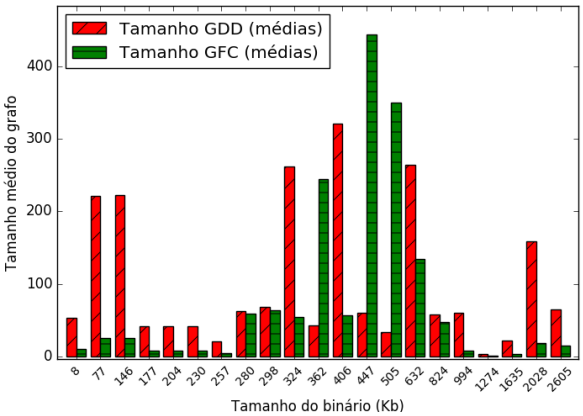


**Figura 4. Tamanhos dos grafos por tamanho dos arquivos**

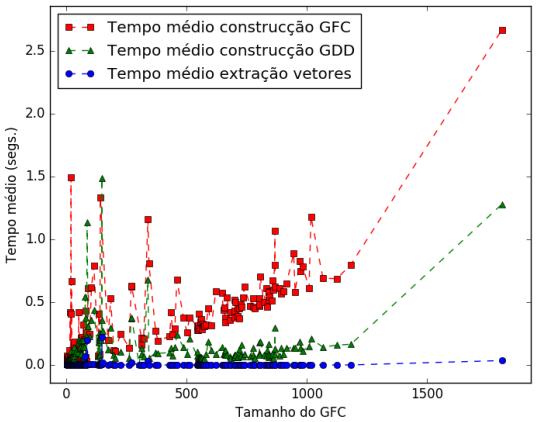


**Figura 5. Tempo médio de construção das assinaturas por tamanho dos GFCs**

cam técnicas de ofuscação como: $a$) Substituição de instruções equivalentes simples e em grupo; $b$) Reordenação de instruções; $c$) Inserção de código lixo; e $d$) Renomeação de variáveis. Devido à diversidade e complexidade das técnicas de ofuscação aplicadas, as instâncias geradas podem ser consideradas altamente metamórficas. No total foram utilizadas 301 instâncias de *malware* das quais foram geradas 1021 amostras metamórficas usando o *Code Pervertor* e 2024 amostras usando o *Revert4*.

## Resultados

### Processo de construção das assinaturas

A Figura 4 mostra o tamanho médio dos GFCs e GDDs por tamanho do arquivo, presentes no conjunto total de arquivos. É possível observar que não existe uma relação linear entre o tamanho dos arquivos e o tamanho do GFC e consequentemente o tamanho dos GDDs. Isto ocorre devido ao processo de engenharia reversa possuir falhas no reconhecimento e reconstrução de trechos do código binário original, e ao uso extensível de funções de bibliotecas externas no código que são descartadas no processo de engenharia reversa.

Como esperado, os GDDs são maiores que os GFCs, pois os nós nos GFCs representam blocos de instruções, enquanto nos GDDs cada instrução da origem a um nó. Entretanto, como pode ser observado na Figura 4, existem casos em que o GFC é maior que o GDD. Isto ocorre devido ao código original estar contaminado com muitas instruções lixo, o que acarreta o aumento na quantidade dos blocos de instruções no GFC. No entanto, estas instruções lixos serão ignoradas no momento da construção dos GDDs.

A Figura 5 apresenta o tempo médio de construção das assinaturas (GFC, GDD e vetores de características) pelo tamanho dos GFCs. Em média o tempo de construção de assinaturas é diretamente proporcional aos tamanhos do GFC. Para GFCs com 1700 nós, o tempo médio de construção de assinatura é de 4,13 segundos. Entretanto, como o tempo de construção do CFG engloba o processo de engenharia reversa, um código grande que consome muito tempo para ser analisado pode dar origem a um GFC pequeno.

 © 2017 Sociedade Brasileira de Computação

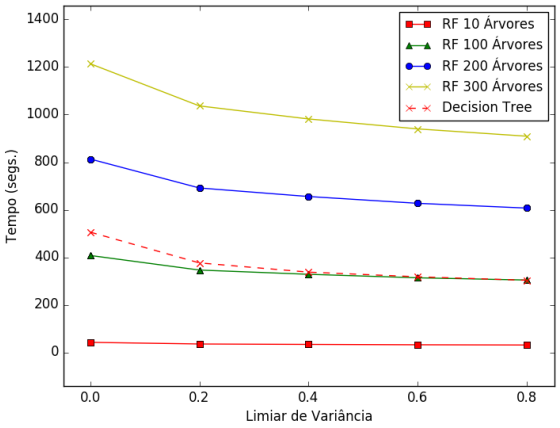


**Figura 6. Tempos de treino dos modelos de classificação.**

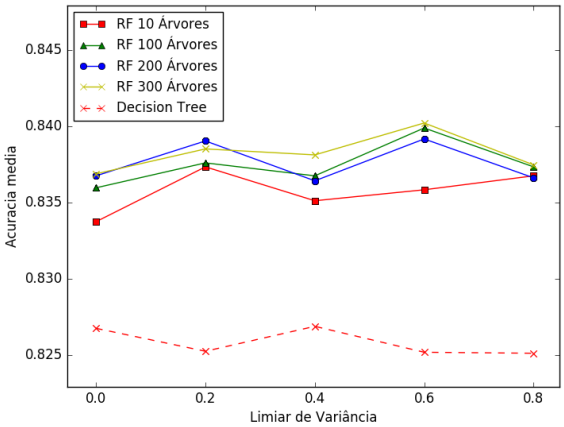


**Figura 7. Acurácia dos modelos de classificação.**

### Avaliação da acurácia dos índices

Para avaliar a acurácia dos índices foram utilizados os seguintes classificadores: uma árvore de decisão C45 e florestas de árvores (*Random Forests*) com 10, 100, 200 e 300 árvores. Os modelos de classificação foram gerados usando 70611 amostras e testados com 30262 amostras. O cálculo da acurácia corresponde ao total de amostras corretamente classificadas sob o total de amostras utilizadas nos experimentos e é obtido pela seguinte fórmula:

$$acuracia(y, y') = \frac{1}{n_{vetores}} \sum_{i=0}^{n_{vetores}-1} 1(y'_i = y_i)$$

A Figura 6 mostra o tempo necessário para gerar os modelos de classificação em função do limiar de redução, obtido a partir da variância das amostras. Nos experimentos realizados foi observado que o tempo de geração dos modelos de classificação diminuia à medida que variância das amostras aumentava. Isto ocorreu pois o nível de variância dos valores das características foi usado como limiar de redução de características que seriam efetivamente usadas na construção do modelo.

A Figura 7 apresenta as acurácias médias dos modelos treinados para os diferentes limiares de variância utilizada para a redução do espaço de características. Os resultados dos experimentos mostram que o processo de redução também não afetou diretamente a acurácia dos modelos de classificação gerados.

### Comparação com antivírus comerciais

Esta seção descreve os resultados da comparação da abordagem proposta com ferramentas comerciais disponibilizadas no sistema *VirusTotal* [Total 2017]. Foram selecionadas 300 amostras previamente identificadas pelos 58 antivírus comercias avaliados para a geração de novas instâncias metamórficas com o uso da ferramenta *Revert4*. No total foram geradas 2186 novas instâncias metamórficas as quais foram submetidas ao *VirusTotal*

**Tabela 3. Acurácia média dos modelos obtidos comparado a antivirus comerciais**

| Posição Ranking | Detector | Acurácia média |
|---|---|---|
| 1 | McAfee-GW-Edition | 88.43 |
| 2 | CrowdStrike | 85.13 |
| 3 | McAfee | 84.26 |
| 4 | **RF N=300 V=0.6** | 84.02 |
| 5 | **RF N=100 V=0.6** | 83.99 |
| 6 | **RF N=200 V=0.6** | 83.92 |
| 7 | Qihoo-360 | 83.81 |
| 8 | Ikarus | 81.75 |
| 9 | Rising | 81.33 |
| 10 | Avast | 80.74 |
| ... | ... | ... |
| 56 | Paloalto | 19.441903 |
| 57 | ClamAV | 15.141812 |
| 58 | Webroot | 13.586459 |

A Tabela 3 apresenta um ranking da acurácia média obtida pelos 10 melhores resultados incluindo os 3 melhores resultados dos modelos de identificação gerados neste trabalho. Também se incluem os três piores resultados do conjunto total, ao final da tabela. Os valores demonstram que a taxa média de acurácia obtida pelos modelos treinados é superior a maioria das ferramentas comercias disponíveis, provando a validade do modelo de identificação proposto.

## Conclusões e Trabalhos Futuros

Este trabalho propôs uma abordagem para a identificação de *malware* resiliente ao metamorfismo de código. Os experimentos mostram (Figura 5) que a abordagem proposta apresenta tempos médios de execução menores aos 5 segundos para a construção das assinaturas utilizadas na detecção. Devido ao tamanho dos grafos não estar relacionado com o tamanho dos arquivos, não existe restrição neste sentido enquanto aos binários que podem ser analisados por esta proposta.

Apesar de estar sendo utilizado para a detecção apenas um GDD por binário analisado, os resultados na acurácia média são competitivos em comparação aos dos antivirus comerciais. Uma abordagem a ser estudada por ser considerada promisora pelos pesquizadores deste trabalho consiste em basear o resultado final da detecção na junção dos resultados da aplicação dos classificadores nos vetores de caracteristicas do conjunto de grafos do binário em análise. Desta forma é esperado o aumento da acuracia e a diminuição dos falsos positivos, aspecto este muito importante na area de detecção de malware.

Próximas etapas deste trabalho incluem o estudo de modelos treinados utilizando outros algoritmos de aprendizagem de maquina assim como a avaliação da qualidade dos classificadores utilizando métricas que considerem o risco de falsos positivos podendo assim também definir estrategias práticas para melhorar a classificação por medio do ajuste dos hiperparametros atendendo á características mais especificas em dependencia do contexto.

 © 2017 Sociedade Brasileira de Computação

## Referências

Ahmadi, M., Sami, A., Rahimi, H., and Yadegari, B. (2013). Malware detection by behavioural sequential patterns. *Computer Fraud & Security*, 2013(8):11–19.

Alam, S., Sogukpinar, I., Traore, I., and Nigel Horspool, R. (2015a). Sliding window and control flow weight for metamorphic malware detection.

Alam, S., Traore, I., and Sogukpinar, I. (2015b). Annotated Control Flow Graph for Metamorphic Malware Detection. *The Computer Journal*, 58(10):2608–2621.

AV-Test (2015). AV-Test 2015 Security Report.

Capstone-Disassembler (2017). Capstone disassembler github repository. `https://github.com/aquynh/capstone`.

Cooper, K. D., Harvey, T. J., and Kennedy, K. (2004). Iterative data-flow analysis, revisited. Technical report.

Eskandari, M. and Hashemi, S. (2012). A graph mining approach for detecting unknown malwares. *Journal of Visual Languages & Computing*, 23(3):154–162.

Ferrante, J., Ottenstein, K. J., and Warren, J. D. (1987). The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349.

Garey, M. R. and Johnson, D. S. (1990). *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA.

Hu, X., Chiueh, T.-c., and Shin, K. G. (2009). Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 611–620. ACM.

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

Kotsiantis, S. B., Zaharakis, I., and Pintelas, P. (2007). Supervised machine learning: A review of classification techniques.

Kuriakose, J. and Vinod, P. (2014). Ranked linear discriminant analysis features for metamorphic malware detection. In *2014 IEEE International Advance Computing Conference (IACC)*, pages 112–117. IEEE.

Kuriakose, J. and Vinod, P. (2015). Unknown metamorphic malware detection: Modelling with fewer relevant features and robust feature selection techniques. *IAENG International Journal of Computer Science*, 42(2):139–151.

Lattner, C. and Adve, V. (2004). LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California.

Lejska, K. (2017). X86 opcode and instruction reference.

Lin, D. and Stamp, M. (2010). Hunting for undetectable metamorphic viruses. *Journal in Computer Virology*, 7(3):201–214.

Malshare (2017). Public repository of malware of the malshare project. `http://malshare.com/about.php`.

Martins, G. B., Santos, P., Danrley, V., Souto, E., and Freitas, R. D. (2016). Identificação de Códigos Maliciosos Metamórficos pela Medição do Nível de Similaridade de Grafos de Dependência. In *Anais do XVI Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais*, pages 296–309.

Munson, M. A. and Caruana, R. (2009). On Feature Selection, Bias-Variance, and Bagging.

Nguyen Anh Quynh, C. (2014). Capstone: next generation disassembly framework. `http://www.capstone-engine.org/BHUSA2014-capstone.pdf`.

Paredes, R. and Chávez, E. (2005). Using the k-nearest neighbor graph for proximity searching in metric spaces. In *International Symposium on String Processing and Information Retrieval*, pages 127–138. Springer.

Rad, B. B., Masrom, M., and Ibrahim, S. (2012). Opcodes histogram for classifying metamorphic portable executables malware. In *2012 International Conference on E-Learning and E-Technologies in Education, ICEEE 2012*, pages 209–213. IEEE.

Radare2 (2017). Radare2 github repository. `https://github.com/radare/radare2`.

Raymond, J. W. and Willett, P. (2002). Maximum common subgraph isomorphism algorithms for the matching of chemical structures. *Journal of computer-aided molecular design*, 16(7):521–533.

Singh, T., Di Troia, F., Corrado, V. A., Austin, T. H., and Stamp, M. (2015). Support vector machines and malware detection. *Journal of Computer Virology and Hacking Techniques*.

Symantec (2017). Symantec 2017 internet security threat report.

Total, V. (2017). Virustotal-free online virus, malware and url scanner. *Online: https://www.virustotal.com/en*.

VXHeaven (2017). Computer virus collection. *URL: http://vxheaven. org/vl. php*.

 © 2017 Sociedade Brasileira de Computação