Improving the detection of metamorphic malware through data dependency graphs indexing

Luis Rojas Aguilera, Eduardo Souto and Gilbert Breves Martins,

***Abstract*—Metamorphism have been successfully used in original malicious code to the creation and proliferation of new malware instances, making them harder to detect. This work presents an approach that identifies metamorphic malware through data dependency graphs comparison. Features are extracted on data dependency graphs to build an index that is used to determine which malware family a suspicious code belongs to. Experimental results on 3045 samples of metamorphic malware showed that our proposed approach obtained accuracy rate higher than most commercial anti-malware tools.**

***Index Terms*—malware, metamorfismo, grafos de dependência, engenharia reversa, aprendizagem de máquina.**

I. 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 [2], 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 [3]. 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.

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 [4]. 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.

Este artigo corresponde a uma versão estendida do artigo:"Detecção de *malware* metamórfico baseada na indexação de grafos de dependência de dados"[1] que recebeu o prêmio de menção honrosa da trilha principal do Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSeg 2017)

Luis Rojas Aguilera e Eduarto Souto, Universidade Federal do Amazonas (UFAM), e-mails: {rojas,esouto}@icomp.ufam.edu.br

Gilbert Breves Martins, Instituto Federal do Amazonas (IFAM), e-mail: gilbert.martins@ifam.edu.br

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 [5] [6] [7]; b) análise das distribuições das ocorrências de sequências de *opcodes* de instruções [8]; c) análise estatística composta pela combinação de métodos de ranqueamento de características de grupos de *opcodes* de instruções [9]; e d) análise de representações intermediárias que expressam as semânticas do código, tais como: grafos de controle de fluxo [10] [11], grafos de chamadas a APIs do sistema [12] e grafos de dependência de dados [13] [4].

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. [14] e Kim e Moon [13], por exemplo, demonstraram que 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 [15].

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 mostram que a abordagem proposta apresenta taxas médias de acurácia superiores a maioria dos antivírus comerciais.

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

II. Trabalhos Relacionados

Em função de sua natureza não trivial, diversas abordagens foram propostas para tratar o problema da identificação de *malware* metamórfico. Essa seção apresenta uma revisão do estado da arte referente a detecção deste tipo de *malware*.

As pesquisas apresentadas foram divididas em dois grupos, de acordo com o enfoque de identificação adotado: *a*) reconhecimento de padrões e *b*) comparação de estruturas semânticas em forma de grafos. O primeiro grupo é composto por abordagens baseadas na utilização de modelos projetados para diferenciar padrões que indiquem uma natureza maligna ou benigna, através da análise de códigos de *malware*, programas livres de contaminação e motores de metamorfismo, também conhecidos como *engines metamórficas*. O segundo grupo é composto pelas abordagens baseadas na comparação exata de estruturas extraídas previamente de instâncias de *malware*, com aquelas presentes em uma instância sob investigação.

A. Reconhecimento de padrões

Os trabalhos de pesquisa apresentados nessa seção baseiam-se na utilização de modelos com a finalidade de determinar se a instância em análise se encaixa em algum padrão previamente obtido de: *a*) famílias de código malicioso, *b*) código benigno, *c*) *engines* de metamorfismo e *d*) kits de geração de *malware*. Em sua maioria, estes métodos utilizam algoritmos de aprendizagem de máquina para construir modelos de classificação capazes de discriminar padrões de comportamentos malignos ou benignos.

Canfora et al. [16] estudaram *engines* de metamorfismo conhecidos e identificaram que era possível levantar padrões da utilização dos *opcodes* pelos *engines* no processo de mutação e integração de código. Baseado nessa premissa, Canfora et al. propõem um sistema de detecção baseada na análise das distribuições de frequência dos *opcodes* em *malware* metamórfico e código benigno. Os autores propõem uma estrutura chamada de Matriz de Ocorrências de Intruções (*Instructions Occurence Matrix* - IOM), na qual são associados cada *opcode* com o número de instruções que o contém, sem considerar *opcodes* que aparecem apenas uma vez.

As IOM, coletadas de instâncias conhecidas como *malware* e código benigno, são utilizadas como vetor de características para gerar classificadores capazes de determinar se uma instância em análise pode ser considerada *malware* ou não. Testes aplicados em 250 instâncias benignas e 500 *malwares* metamórficos, gerados pelo uso dos kits de geração de vírus G2, MPCGEN, NGVCK, NRLG e SMEG, apresentaram uma taxa média de acurácia de 94% e uma taxa média de falsos positivos de 2%. Entretanto, testes posteriores envolvendo instâncias metamórficas geradas a partir de código benigno, na tentativa de identificar o código da instância original não maliciosa, obtiveram taxas de acurácia entre 75% e 92%.

Choudhary e Vidyarthi [17] propõem uma abordagem de detecção baseada no reconhecimento de padrões no rastreamento do fluxo de execução de *malware* e código benigno. Os programas são executados em ambiente virtualizado, com o intuito de criar um relatório com informações que caraterizem o comportamento do binário a partir das: *a)* operações executadas pelos processos e subprocessos iniciados por este; *b)* entradas do registro do sistema que são afetados; e *c)* valores dos parâmetros utilizados por estes processos. Em seguida, técnicas de mineração de dados são empregadas sobre os relatórios gerados para determinar as distribuições de frequência e ganho de informação de seus elementos textuais. Para cada elemento contido no texto, é obtido o coeficiente da probabilidade deste elemento aparecer em relatórios gerados a partir de instâncias benignas ou malignas. Estes coeficientes de probabilidades são a base para a construção dos modelos de classificação. Os experimentos mostram que os testes com 188 instâncias, das quais 91 eram malignas, apenas 4 instâncias foram classificadas incorretamente. No entanto, a abordagem é vulnerável a técnicas de anti-análise incorporadas em *malware* capazes de detectar quando o código está sendo executado em ambientes virtuais, podendo fazer com que este apresente um comportamento não suspeito para evitar sua detecção.

B. Comparação de estruturas semânticas em forma de grafos

Nesta seção são apresentados trabalhos que baseiam o processo de detecção na comparação das instâncias suspeitas com outras previamente identificadas por meio da utilização de estruturas extraídas do corpo do código. A utilização de estas estruturas tem como objetivo fazer o método de detecção resiliente ao metamorfismo de código, pois representam a intenção e funcionalidade do código a partir dos fluxos de dados e controle presentes no mesmo.

Xin Hu et al. [10] propõem 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*) [18]. 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 [13] 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 [19]. 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. [4] exploraram esta limitação, introduzindo a classificação de nós baseado na semântica e função das instruções correspondentes. A análise foi feita dividindo os 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 mostram uma diminuição tanto no tamanho dos grafos manipulados, como na variância dos resultados obtidos no processo de comparação.

Eskandari and Hashemi [20] 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 nesses 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 entre os que se encontram *Decision Stump*, *Decision Tree*, *Random Forest*, *Naive Bayes*, entre outros. O modelo de classificação treinado utilizando *Random Forest* apresentou a melhor acurácia de 81%.

Alam et.al [21] 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 código 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*.

III. 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 [21]:

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) [22] 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 em particular depende, este trabalho adota uma definição para arestas de dependência adaptada daquela apresentada em Kim e Moon [13]:

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.

A. 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.

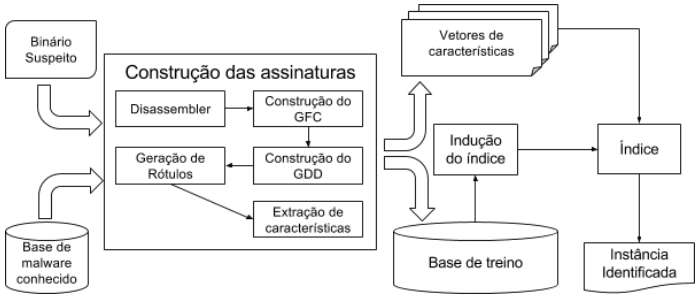

Fig. 1. Visão geral da abordagem proposta para detecção de *malware* metamórficos.

B. 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 [23], 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 $worklist$ [24].

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.

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
```

C. 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 metamórficas podem ser criadas.

TABLE I
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
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

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 [25]. 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*) [26], também adotada pelo disassembler Capstone [27] [28].

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

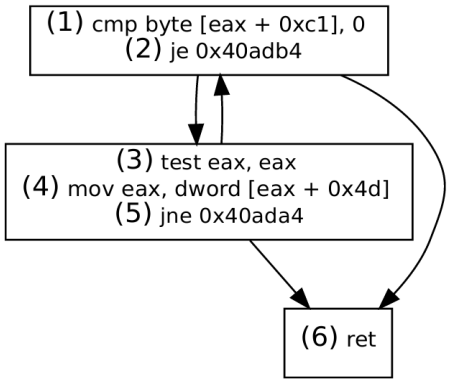

Fig. 2. Exemplo de um GFC

A Tabela II 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*.

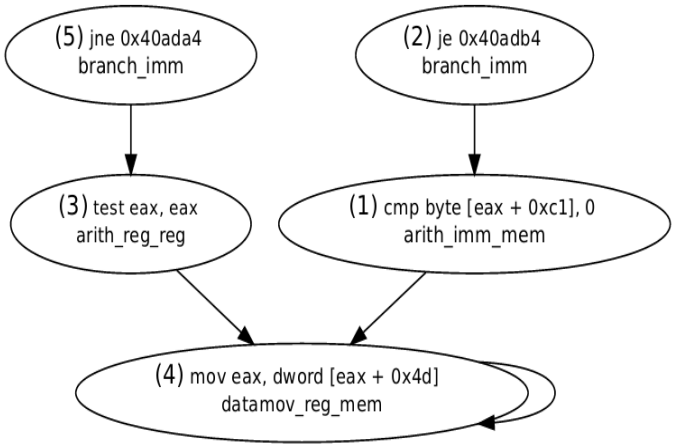

Fig. 3. GDD rotulado extraído do GFC na Figura 2

TABLE II
EXEMPLO DO VETOR DE CARACTERÍSTICAS OBTIDO DO GDD NA FIGURA 3.

n_branch_imm	i_branch_imm	o_branch_imm	n_arith_reg_reg	...
2	0	2	1	...

D. 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 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 [29]. 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

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

TABLE III
DATASET DISTRIBUTION

	Malware samples	**Avg. File Size / Standard Deviation**	**Graphs extracted**
Seeds	291	816Kb / 258	8825
Code pervertor	3574	965Kb / 395	43975
Revert4	665	871Kb/ 357	48073

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 [30]. 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.

IV. EXPERIMENTAÇÃO E RESULTADOS

A avaliação da abordagem proposta foi baseada em um conjunto de experimentos. Devido ao dataset de vetores de características utilizado apresentar um tamanho superior aos 8GB quando carregado em memória e para aproveitar a capacidade de processamento 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.

A. 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* [31], 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* [32]. Estas ferramentas aplicam 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*.

B. Resultados

1) 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 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.

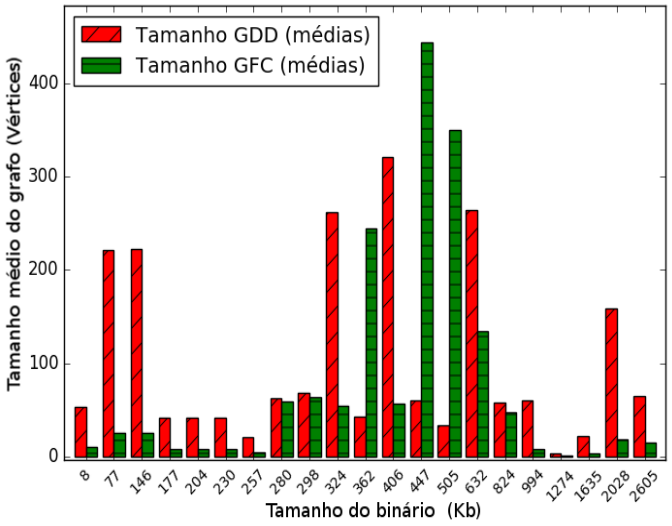

Fig. 4. Tamanhos dos grafos por tamanho dos arquivos

Na Figura 5 são plotados os tempos médios de *dissasembling* para diferentes faixas de tamanhos de arquivos, sendo cada curva referente a uma categoria de mutação distinta das presentes neste trabalho: sementes(seeds), instâncias geradas com o mutador Code Pervertor e com o mutador Revert4.

Quando versões metámórficas dos binários são geradas pelo uso do *Code Pervertor*, o tamanho original do binário não é afetado. No caso do *Revert4*, ao serem aplicadas técnicas avançadas, por meio da integração de código, as instâncias metamórficas geradas possuem tamanhos diferentes daqueles encontrados nas instâncias sementes. No entanto, nas faixas de tamanho de arquivo selecionados na Figura 5 estão contidas instâncias pertencentes a mesma família.

Em média, os tempos de *disassembling* são maiores para instâncias mutadas do que aqueles referentes as instâncias sementes. Como explicado por *Z0mbie* [33], as modificações aplicadas pelos mutadores tem também como objetivo ludibriar o processo de engenharia reversa por meio da integração

de código lixo, usualmente carregado de instruções de controle que resultam em fluxos de execução complexos e difíceis de reconstruir pelas técnicas de engenharia reversa. Entre as técnicas utilizadas para este fim, encontram-se: a) inserção de código lixo; b) randomização da distribuição das instruções e dados; e c) código entrelaçado (código espaguete). Estas modificações impactam no processo de identificação de referências cruzadas. Tal processo é muito importante para construir um mapa semântico mais preciso, o que é necessário para reconstruir a uma representação de alto nível mais fiel ao código binário original. Apesar das técnicas utilizadas incorporarem alta complexidade ao processo, para a maioria dos arquivos, o tempo de *disassembling* é inferior a cinco segundos.

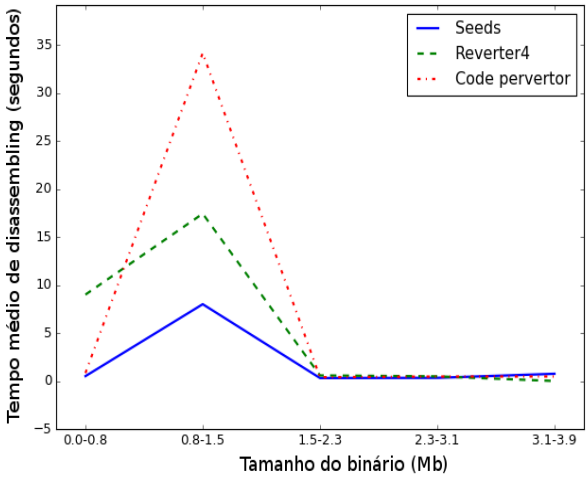

Fig. 5. Tempo médio de disassembling por faixas de tamanhos de arquivo presentes no dataset.

As Figuras 6, 7 e 8 apresentam o tempo que a implementação da arquitetura proposta levou para construir os grafos (i.e GFC e GDD) por faixas de tamanhos de GFC presentes no dataset, para instâncias sementes, geradas usando *Code Pervertor* e *Revert4*, respetivamente. O tempo de execução dos algoritmos utilizados para gerar estes grafos aumentam proporcionalmente ao tamanho dos grafos, razão pela qual as curvas referentes a cada grafo apresentam um crescimento similar. Para a maioria das faixas de tamanho selecionados dos GFCs, os valores de tempo de construção dos GDDs são próximos aos dos GFCs.

Como as transformações aplicadas pelo *Code Pervertor* não afetam o tamanho e a forma do GFC, os tempos, formas das curvas e faixas de tamanho dos GFCs nas Figuras 7 e 6 são similares. No entanto, para GFCs com mais de 240 nós, os tempos para extração dos grafos de instâncias geradas com *Code Pervertor* são menores. Isto ocorre devido as mudanças no código introduzidas pelos mutadores, uma análise mais detalhada mostrou que algumas instruções e blocos básicos acabaram não sendo analisadas pelo algoritmo de extração.

Na Figura 8 pode ser visto como os tamanhos dos GFCs são maiores para as instancias geradas com *Revert4*. Isto ocorre devido ao extensivo uso de técnicas de metamorfismo que

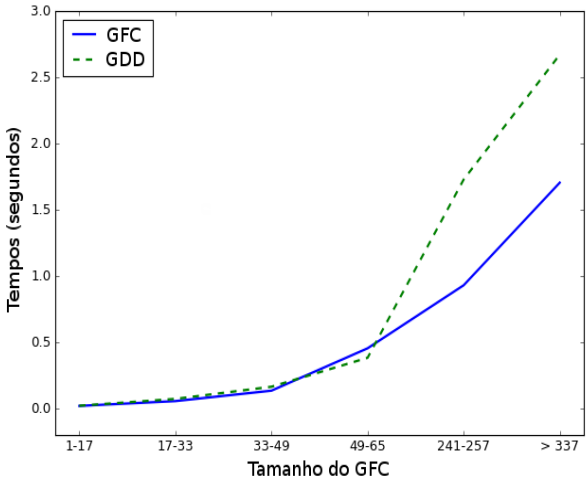

Fig. 6. Tempo médio de construcção dos GFC e GDD por tamnho do GFC em arquivos sementes.

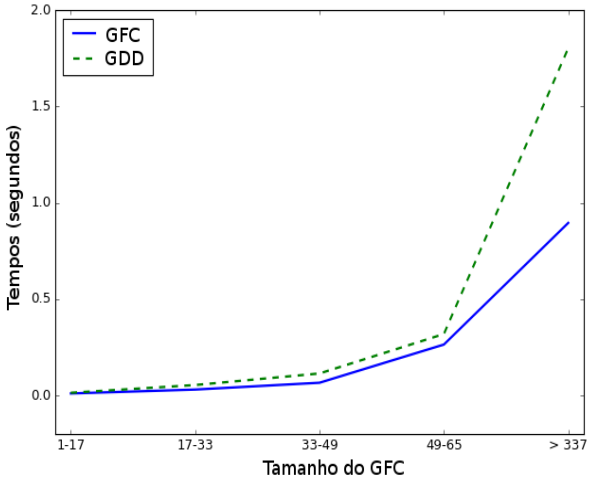

Fig. 7. Tempo médio de construcção dos GFC e GDD por tamnho do CFG em arquivos mutados com Code Pervertor.

atuam no fluxo de controle, como código *espaguete* e código lixo, o que resulta na inclusão de novos blocos básicos nos GFCs. Isto também afeta o tempo de processamento, devido ao aumento dos blocos básicos no GFC que tem de ser processados tanto pelo *disassembler*, para a construção dos GFCs, como pelo algoritmo de análise do fluxo de dados, para a construção dos GDDs.

2) Avaliação da acurácia dos índices: Para avaliar a acurácia dos índices foram utilizados os seguintes classificadores: *a)* uma árvore de decisão C45 e *b)*florestas de árvores (*Random Forests*) com 10, 100, 200 e 300 árvores. Os modelos de classificação foram gerados usando 70611 grafos 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:

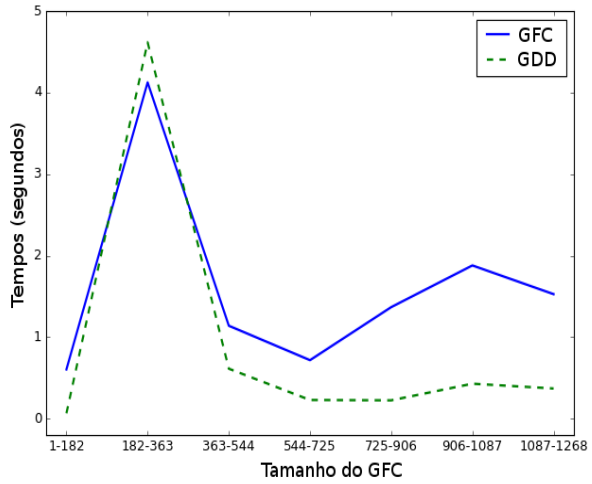

Fig. 8. Tempo médio de construcção dos CFG e DDG por tamnho do CFG em arquivos mutados com Revert4.

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

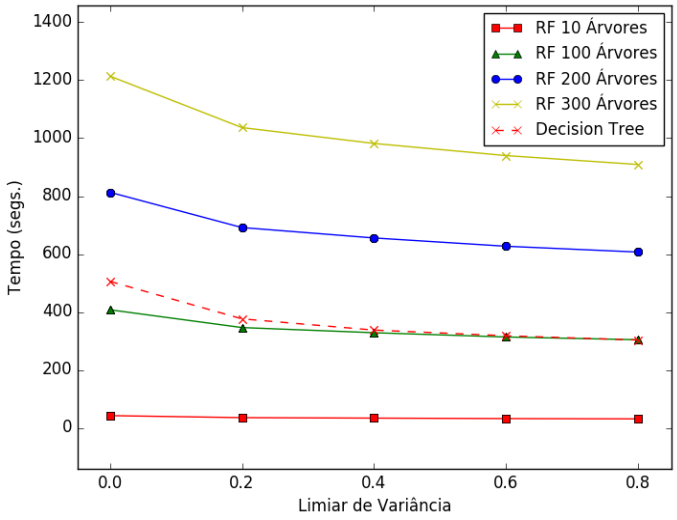

Fig. 9. Tempos de treino dos modelos de classificação.

A Figura 9 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 10 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

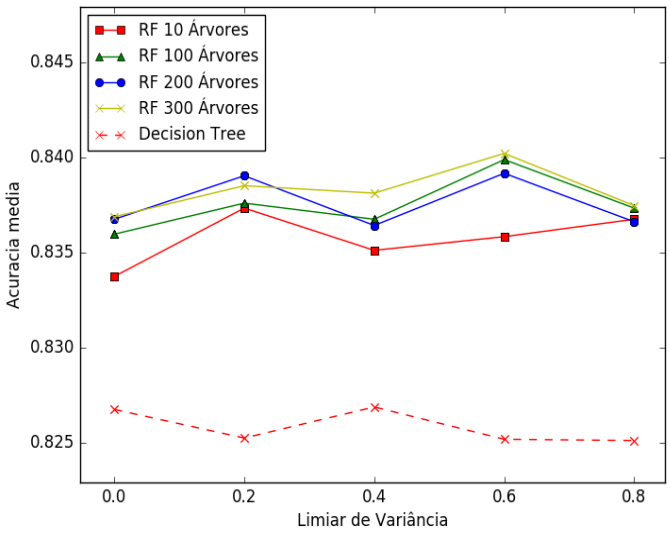

Fig. 10. Acurácia dos modelos de classificação.

TABLE IV
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

afetou diretamente a acurácia dos modelos de classificação gerados.

3) Comparação com antivírus comerciais: Esta seção descreve os resultados da comparação da abordagem proposta com ferramentas comerciais disponibilizadas no sistema *VirusTotal* [34]. Foram selecionadas 300 amostras previamente identificadas pelos 58 antivírus comerciais 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*

A Tabela IV apresenta um ranking da acurácia média obtida pelos 10 melhores resultados incluindo os 3 melhores resultados dos modelos de identificação gerados neste trabalho. 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.

V. CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho propôs uma abordagem para a identificação de *malwares* metamórficos. Os experimentos mostraram (Figuras 6, 7 e 8) que a abordagem proposta apresenta tempos médios

de execução inferiores a 5 segundos. Além disso, apesar de estar sendo utilizado somente um GDD por binário analisado no processo de detecção, os resultados na acurácia média são competitivos em comparação aos resultados obtidos pelos antivirus comerciais. As próximas etapas deste trabalho incluem o estudo de modelos treinados utilizando outros algoritmos de aprendizagem de máquina, bem como a avaliação da qualidade dos classificadores utilizando métricas que considerem as taxas de falsos positivos.

REFERENCES

[1] L. M. Rojas, E. Souto, and G. Breves, "Deteccao de malware metamorfico baseada na indexacao de grafos de dependencia de dados," in *XVII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais: SBSEG 2017: Anais*, pp. 264–277, SBC, 2017.

[2] AV-Test, "AV-Test 2015 Security Report," 2015.

[3] Symantec, "Symantec 2017 internet security threat report," 2017.

[4] G. B. Martins, P. Santos, V. Danrley, E. Souto, and R. D. Freitas, "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*, pp. 296–309, 2016.

[5] D. Lin and M. Stamp, "Hunting for undetectable metamorphic viruses," *Journal in Computer Virology*, vol. 7, pp. 201–214, dec 2010.

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

[7] J. Kuriakose and P. Vinod, "Ranked linear discriminant analysis features for metamorphic malware detection," in *2014 IEEE International Advance Computing Conference (IACC)*, pp. 112–117, IEEE, 2 2014.

[8] B. B. Rad, M. Masrom, and S. Ibrahim, "Opcodes histogram for classifying metamorphic portable executables malware," in *2012 International Conference on E-Learning and E-Technologies in Education, ICEEE 2012*, pp. 209–213, IEEE, sep 2012.

[9] J. Kuriakose and P. Vinod, "Unknown metamorphic malware detection: Modelling with fewer relevant features and robust feature selection techniques," *IAENG International Journal of Computer Science*, vol. 42, no. 2, pp. 139–151, 2015.

[10] X. Hu, T.-c. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proceedings of the 16th ACM conference on Computer and communications security*, pp. 611–620, ACM, 2009.

[11] S. Alam, I. Sogukpinar, I. Traore, and R. Nigel Horspool, "Sliding window and control flow weight for metamorphic malware detection," 2015.

[12] M. Ahmadi, A. Sami, H. Rahimi, and B. Yadegari, "Malware detection by behavioural sequential patterns," *Computer Fraud & Security*, vol. 2013, pp. 11–19, 8 2013.

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

[14] C. Liu, C. Chen, J. Han, and P. Yu, "GPLAG: detection of software plagiarism by program dependence graph analysis," *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 872–881, 2006.

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

[16] G. Canfora, A. N. Iannaccone, and C. A. Visaggio, "Static analysis for the detection of metamorphic computer viruses using repeated-instructions counting heuristics," *Journal of Computer Virology and Hacking Techniques*, vol. 10, pp. 11–27, 9 2013.

[17] S. Choudhary and M. D. Vidyarthi, "A Simple Method for Detection of Metamorphic Malware using Dynamic Analysis and Text Mining," *Procedia Computer Science*, vol. 54, pp. 265–270, 2015.

[18] R. Paredes and E. Chávez, "Using the k-nearest neighbor graph for proximity searching in metric spaces," in *International Symposium on String Processing and Information Retrieval*, pp. 127–138, Springer, 2005.

[19] J. W. Raymond and P. Willett, "Maximum common subgraph isomorphism algorithms for the matching of chemical structures," *Journal of computer-aided molecular design*, vol. 16, no. 7, pp. 521–533, 2002.

[20] M. Eskandari and S. Hashemi, "A graph mining approach for detecting unknown malwares," *Journal of Visual Languages & Computing*, vol. 23, no. 3, pp. 154–162, 2012.

[21] S. Alam, I. Traore, and I. Sogukpinar, "Annotated Control Flow Graph for Metamorphic Malware Detection," *The Computer Journal*, vol. 58, pp. 2608–2621, 10 2015.

[22] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[23] Radare2, "Radare2 github repository." https://github.com/radare/radare2, 2017.

[24] K. D. Cooper, T. J. Harvey, and K. Kennedy, "Iterative data-flow analysis, revisited," tech. rep., 2004.

[25] K. Lejska, "X86 opcode and instruction reference," 2017.

[26] C. Lattner and V. Adve, "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), Mar 2004.

[27] C. Nguyen Anh Quynh, "Capstone: next generation disassembly framework." http://www.capstone-engine.org/BHUSA2014-capstone.pdf, 2014.

[28] Capstone-Disassembler, "Capstone disassembler github repository." https://github.com/aquynh/capstone, 2017.

[29] M. A. Munson and R. Caruana, "On Feature Selection, Bias-Variance, and Bagging," 2009.

[30] S. B. Kotsiantis, I. Zaharakis, and P. Pintelas, "Supervised machine learning: A review of classification techniques," 2007.

[31] Malshare, "Public repository of malware of the malshare project." http://malshare.com/about.php, 2017.

[32] VXHeaven, "Computer virus collection," *URL: http://vxheaven. org/vl. php*, 2017.

[33] "Z0MBiE's HomePage."

[34] V. Total, "Virustotal-free online virus, malware and url scanner," *Online: https://www.virustotal.com/en*, 2017.