Mapper - Um Gerador Eficiente de Grafos de Comportamento para Detecção de Malwares

Vinicius Zavala, Magnum Adelano, Brivaldo A. Silva Junior[1]

[1]Faculdade de Computação (FACOM)
Universidade Federal de Mato Grosso do Sul (UFMS)
CEP – 79070-900 – Campo Grande – MS – Brasil

{vinicius.queiroz, magnum.faria}@aluno.ufms.br, brivaldo@facom.ufms.br

Abstract. *Current antivirus use different approaches to detect polymorphic malwares. One way to do this is to map the software behavior to determine whether it is malicious. This processes use as information the main operations of an application. This is necessary as some malwares uses obfuscation techniques to make the signature detection process more difficult and, to bypass the protection tools. Unlike previous approaches, this work uses the read and write processes in memory as dependencies (variables, registers, disk) to determine the depedency degree between calls within a program to create its dependency graph. Experiments show that the new dependency graph is up to 63% lower than the current and still retains the necessary properties required by the existing heuristic algorithms. This graph reduction reduces the memory usage and processing time when evaluation of new variants of a malware is made.*

Resumo. *Os antivírus atuais usam abordagens diferentes na detecção de malwares polimórficos. Uma delas é mapear o comportamento de um software e determinar se ele é ou não malicioso baseado em suas operações mais importantes. Isso é necessário pois muitos malwares usam técnicas de ofuscação para tornar a detecção por assinatura mais difícil e, com isso, conseguir passar despercebido pela ferramenta de proteção. Diferente de abordagens anteriores, este trabalho usa as dependências de leitura e escrita em memória (variáveis, registradores, disco) para determinar o grau de dependência entre chamadas dentro de um programa e criar o seu grafo de dependência. Experimentos mostram que o novo grafo de dependência inicial é até 63% menor que os atuais e ainda mantém as propriedades necessárias para o uso dos algoritmos heurísticos já existentes. Essa redução na entrada reduz o uso de memória e tempo de processamento na avaliação de novas variantes de um malware.*

1. Introdução

Nas últimas décadas, o número de softwares maliciosos aumentou consideravelmente. Dados do relatório de 2013 da McAfee [Report 2014], mostram que ocorreram investimentos na ordem de $300 bilhões e perdas decorrentes de atividades maliciosas com operações na Internet na ordem de $1 trilhão de dólares.

Os *malwares*, como são conhecidos esses *softwares*, são aplicativos que tem como objetivo danificar, roubar ou controlar sistemas. Em resumo, *malware* foi um termo criado para designar qualquer *software* malicioso, seja ele um vírus, um *spyware*, um *worm*, etc.

Este aumento no número de *malwares* se deve ao crescimento do número de dispositivos – *notebooks*, *smartphones*, *tablets*, *video-games*, televisores – interconectados à rede e a chegada de uma nova gama de dispositivos (IoT - *Internet of Things*), o que aumentou a troca de informações e a quantidade de novas formas de interação (novas aplicações, novos problemas, novos protocolos, etc.).

Antigamente, para detectar novos malwares (ou modificações de um mesmo malware), o uso de técnicas de detecção de plágio como as usadas em ferramentas como Moss [Schleimer 2003] e JPlag [Prechelt et al. 2000] eram suficientes para captar as pequenas mudanças no código de um *malware* e atribuir um grau de semelhança entre a praga inicial e a modificada. Isso era ainda mais simples pelo fato de não existirem, na época, ferramentas geradoras e os *malwares* serem modificados manualmente. Contudo, os desenvolvedores dos programas maliciosos aperfeiçoaram seus mecanismos e técnicas de programação, usando meios para dificultar a detecção de suas criações [Bishop 2004]. Técnicas como ofuscação de código, polimorfismo, metamorfismo e, até mesmo, compactação de binários (usando ferramentas como o upx [UPX 2015]), adicionaram uma grande complexidade ao problema e, com isso, mais trabalho às ferramentas de detecção. O trabalho de [Grégio 2012] explica com profundidade como funcionam os principais mecanismos comportamentais de um *malware*.

Trabalhos mais recentes propuseram formas para modelar o comportamento dos *malwares* e suas variantes de maneira eficaz mas com alto consumo de memória. [Cozzolino et al. 2012], [Kolbitsch et al. 2009], [Kim and Moon 2010], [Kim and Moon 2013] e [Choi et al. 2014] avaliaram mecanismos para detectar variantes usando grafos direcionados e técnicas clássicas usadas em isomorfismo de subgrafos. O trabalho de Kim e Moon, por exemplo, utiliza algoritmos genéticos para determinar se o grafo de um *malware* conhecido e um grafo previamente extraído de um arquivo em avaliação são subgrafos isomorfos e, por consequência, o segundo é uma variante do primeiro.

A constante busca por soluções de detecção para as variantes de *malwares* está relacionada com a incapacidade de detecção de uma nova ameaça por antivírus baseados em assinaturas. Na realidade, a nova ameaça nada mais é do que uma antiga mas com uma assinatura (muitas baseadas em técnicas de *hash*) diferente. O trabalho de [Kolbitsch et al. 2009] mostra que é possível extrair de um programa um grafo de comportamento que composto por suas características principais. O primeiro problema é como construir o grafo e o segundo é como determinar, de forma eficaz, que o grafo de um novo arquivo é similar ao de um *malware* conhecido. A solução da segunda parte do problema usa heurísticas na maioria das abordagens pois está relacionado com o isomorfismo de subgrafos que é um problema NP-completo.

Este trabalho propõe uma nova estratégia de mapeamento para construção do grafo de comportamento de uma aplicação. Essa estratégia acelera o processo total de detecção pois reduz o uso de memória necessário do grafo inicial usado, sem degenerar os resultados do comportamento essencial da aplicação.

Para simplificar o processo, um parser da linguagem C é usado para determinar e retirar os *tokens* desejados do código fonte. Dessa forma, com os *tokens* separados, o algoritmo de criação do grafo determina as dependências e cria o grafo de comportamento da aplicação.

O restante do artigo está organizado da seguinte forma: na Seção 2, são discutidos os trabalhos relacionados, apresentando diversas tecnologias utilizadas na identificação de códigos

polimórficos. A Seção 3 apresenta a implementação do algoritmo usado no *Mapper*. Na Seção 4, é feita uma análise dos resultados obtidos. Finalmente, na Seção 5, são apresentadas as conclusões finais e propostas de trabalhos futuros.

2. Trabalhos Relacionados

A detecção de *malwares* polimórficos (que tem o mesmo comportamento malicioso mas com código modificado) tem sido abordada por diversas estratégias. A maioria dos trabalhos utiliza grafos para construir mapas computacionais do comportamento de um software e, a partir dai, algoritmos e ferramentas para determinar se um programa é ou não similar a um programa malicioso já conhecido.

O trabalho de [Cozzolino et al. 2012] utiliza a normalização de código para construir blocos comportamentais (*tokens*) que são delimitados por instruções de código de salto. Esses blocos são usados na criação de identificadores de um fluxo que servem para comparações futuras. Dessa forma, uma nova aplicação analisada é maliciosa se o seu fluxo for similar ao fluxo de um *malware* previamente identificado. O processo que determina se uma aplicação está ou não infectada é baseada na quantidade do número de fluxos similares.

Se o número de fluxos similares for alto (processo que exige, algumas vezes, vários passos de normalização no mesmo código), o novo arquivo será marcado como infectado. Essa heurística, quando ocorrem falsos positivos, depende de interação humana. Isso pode impactar na usabilidade do mecanismo em ferramentas de detecção automatizada.

[Kolbitsch et al. 2009] *et al* não utiliza fluxos para mapear o comportamento de uma aplicação. Seu funcionamento é baseado na relação entre chamadas de sistema para construir um grafo acíclico dirigido (DAG) de comportamento baseado no fluxo das informações dentro de um Sistema Operacional (OS). Esse processo ocorre dentro de um *sandbox* virtual (QEMU) que roda o Anubis (uma ferramenta capaz de mapear todas as chamadas de uma aplicação em análise dentro da jaula) que extrai, de fato, todas as informações necessárias do sistema em um formato de marcações.

O problema da estratégia utilizada é que todas as aplicações em análise devem ser previamente enviadas ao *sandbox* para que sejam classificadas como livres ou infectadas. Além do processo de envio e classificação tornar o modelo mais lento em termos de tempo de resposta, o ambiente precisa ser pré-configurado. Isso é importante, pois em uma das fases da ferramenta é realizada a execução do processo de *slice* na aplicação. Esse processo consiste no correto mapeamento das variáveis de entrada em uma chamada de função com os seus valores de saída (dados coletados do Anubis). Dessa forma, com esses dados mapeados, é construído o grafo de comportamento da aplicação. O grafo gerado é então comparado com uma base de conhecimento que determina se a aplicação analisada é maliciosa ou não, mas esse processo de avaliação não é completamente descrito no trabalho.

O uso de grafos direcionados no mapeamento do comportamento de um programa se mostrou uma forma simples e eficiente na detecção de *malwares* polimórficos. Outro trabalho que usou essa estratégia foi o de [Kim and Moon 2010]. Contudo, a geração de grafos, quando não são utilizados mecanismos de mapeamento em nível de linguagem de máquina, dependem do código fonte ou algo similar do *malware*. No caso, as aplicações maliciosas analisadas por [Kim and Moon 2010] eram escritas nas linguagens Visual Basic Script e JavaScript.

O trabalho deles se destaca em dois aspectos. O primeiro é relacionado a criação de

um grafo de comportamento do *malware* baseando-se nas linhas de código, conversões de instruções de laço de repetição em comparações diretas (laços `for` e `while` em estruturas `if` e `goto`), a relação de dependência entre as linhas e um processo de poda para remover estruturas desnecessárias. E o segundo, em relação ao uso de um algoritmo genético para determinar se dois grafos (aplicação em análise e *malware*) são subgrafos isomorfos. O isomorfismo de subgrafo (problema NP-Completo) foi utilizado para determinar se o grafo da aplicação em análise era ou não similar ao grafo de um *malware* conhecido. Dessa forma, com um certo limiar, uma nova aplicação era ou não definida como uma variação de uma aplicação maliciosa já conhecida.

Um dos problemas do trabalho de [Kim and Moon 2010] é que o grafo de entrada é muito grande. Isso se deve ao fato de que a geração do grafo é baseada nas linhas do programa. Outro agravante da estratégia baseada em linhas, é que após a conversão dos laços em outras estruturas de decisão, o tamanho do código (em número de linhas) aumenta mais ainda. Esse grafo é o mesmo usado por outros trabalhos recentes como os de [Choi et al. 2012], [Choi et al. 2014] e [Martins et al. 2014]. Dessa forma, ambos os trabalhos relacionados enfrentam o mesmo problema.

3. Implementação

O Mapper é implementado em Python usando principalmente as bibliotecas [Network 2015] e [PyCParser 2015]. O NetworkX serviu para a criação do grafo direcionado e o PyCParser para extrair os *tokens* dos programas escritos em ANSI C. Além disso, a flexibilidade da linguagem auxiliou na agilidade no desenvolvimento da aplicação e na possibilidade de expandir o trabalho para experimentar implementações referentes a isomorfismo de subgrafo usando técnicas clássicas ou algoritmos genéticos. Todo o processo de validação do algoritmo foi executado em um computador rodando Debian Linux 7 com 4GB de RAM e processador de 2.8GHz.

A aplicação que implementa o algoritmo de criação do grafo de dependências do Mapper usa um parser de linguagem C para extrair os *tokens* de um código fonte e, a partir dessas informações, construir os vértices e arestas do grafo. O grafo é direcionado mas pode conter ciclos. Cada vértice do grafo representa uma variável em memória ou então uma modificação. As arestas do grafo representam a dependência. Toda operação unária, binária ou ternária cria uma dependência entre a variável lida e a modificada. A Figura 1 mostra um exemplo de um grafo de dependência entre a variável X e Y. Um grafo de comportamento é composto por um conjunto de grafos de dependência.

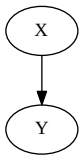

Figura 1. Grafo da relação entre duas variáveis X e Y em um código.

Embora a implementação tenha usado um parser para a linguagem C, o algoritmo independe da linguagem em si. Ou seja, um parser para *Assembly* ou qualquer outra linguagem poderia ser utilizado para realizar a construção do grafo de comportamento. Além disso, a linguagem C é comum em *malwares* por ser extremamente robusta e poderosa.

```
int fat, n;
for(fat = 1; n > 1; n--)
  fat = fat * n;
return fat;
```

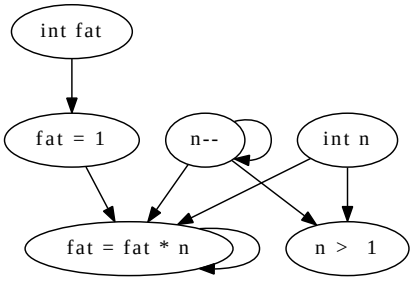

Figura 2. Implementação do algoritmo fatorial sem modificações e o seu grafo de comportamento.

Os *malwares* polimórficos recebem esse nome por utilizarem técnicas de ofuscação de código para evitar a detecção por softwares de proteção. Para demonstrar a implementação, o algoritmo do fatorial foi utilizado e modificado com técnicas polimórficas distintas. A Figura 2 apresenta o código do fatorial original (sem modificações) e o seu grafo de comportamento, a Figura 3 usa as técnicas de renomeação e reordenação de variáveis e a Figura 4 a inserção de lixo no código.

```
int index;
int fact;
for(fact=1; index>1; index--)
  fact = index * fact;
```

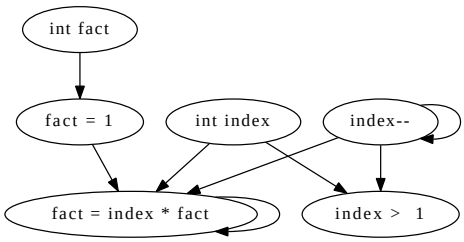

Figura 3. Implementação do algoritmo fatorial aplicando as técnicas de polimorfismo de reordenação e renomeação de variáveis.

Os três grafos de dependência apresentados nas figuras são distintos entre si. Os dois primeiros são similares, pois as variáveis foram apenas trocadas e reordenadas, o que criou grafos com a disposição dos vértices diferente. Contudo, essa modificação já alteraria o *hash* de uma assinatura para o *malware* original, confundindo uma ferramenta baseada apenas em assinaturas. A terceira figura apresenta um grafo totalmente diferente dos anteriores pois agora temos várias entradas adicionais (lixo).

```
int fat;
int n; char prod;
float junk, trash = 3;
fat = 2;
prod = 'd';
junk = 99.9;
for(fat = 1; n > 1; n--)
fat = fat * n;
trash = 160;
```

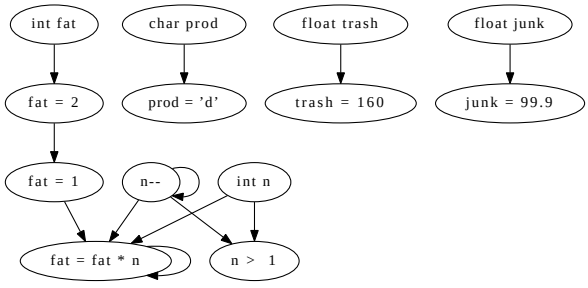

Figura 4. Implementação do algoritmo fatorial aplicando as técnica de polimorfismo de inserção de lixo.

O processo de construção do grafo de comportamento é diferente do apresentado por [Kim and Moon 2010]. Na implementação de [Kim and Moon 2010], um código fonte é

recebido como entrada e um grafo de comportamento é retornado. O processo de geração deste grafo depende de algumas etapas. Primeiro, o código fonte é recebido e convertido em um código com significado semântico. Após essa conversão, que aumenta ainda mais o número de linhas do código, é criado um grafo de fluxo de controle das linhas do código. Usando esse grafo de fluxo, é criado então o grafo de comportamento.

Algoritmo 1: Mapeamento de variáveis para o grafo de comportamento.

Entrada: Código em ANSI C da Aplicação em fase de análise
Saída: Grafo de Conjunto de Vértices

```
1 início
2     Seja Vértices[], uma pilha de todos os vértices criados no grafo;
3     Seja T o conjunto de todos os (tokens) do código suspeito analisado;
4     para todo x ∈ T, faça
5         se x ∈ {variável ou operações(binária, unária, ternária, atribuição)} então
6             Crie um vértice com rótulo x;
7             Adicione x a Vértices[];
8 fim
```

O Mapper, diferente da implementação de [Kim and Moon 2010], analisa diretamente a relação de dependência de leitura e escrita de variáveis em memória para criar diretamente o grafo de comportamento. Esse mapeamento é feito usando o Algoritmo 1 que extraí o conjunto de dados mapeados como vértices. Em seguida, a saída do Algoritmo 1 é usada como entrada para o Algoritmo 2 que determina a relação de dependência entre duas variáveis no grafo e retorna o grafo de comportamento gerado.

O processo de construção do grafo de comportamento, seguindo o algoritmo, recebe um código fonte compilável em C. Ao receber esse código, o PyCParser realiza a análise sintática linha a linha do código fonte e extraí todas as informações do código. Além disso, foram criadas duas estruturas de dados (uma lista e uma pilha) para armazenar as variáveis e estruturas de controle (`if`, `for`, `while`, etc.) de interesse. Um vértice é criado no grafo seguindo algumas regras:

1. Quando uma variável é declarada;
2. Para todas as atribuições;
3. Para todas as operações unárias, binárias e ternárias.

Durante a criação do grafo de comportamento, os laços de controle, como o laço `for`, devem ser tratados de forma especial. No caso do laço `for`, por exemplo, sua entrada é divida em três: atribuição, condição e incremento. Cada uma delas é convertida em um vértice no grafo (assim como todas as operações unárias, binárias ou ternárias do código). E o corpo do código segue as regras definidas no algoritmo para criação das arestas. Outras estruturas de controle como o `while` ou `do..while` podem ser considerados como casos especiais do laço `for`.

4. Resultados Obtidos

Para validar o algoritmo da Seção 3 foram criados vários *malwares* fictícios (sem capacidade de causar danos reais). Esses programas foram modificados para manter a estrutura da aplicação

Algoritmo 2: Gerador do grafo de comportamento final pronto para redução.

Entrada: Conjunto de Vértices gerados no Algoritmo 1
Saída: Grafo de Comportamento

```
1  início
2      Seja A o conjunto de todas as atribuições do código suspeito;
3      para todo x ∈ A, faça
4          para todo y ∈ Vértices[], faça
5              se y ⊂ x então
6                  Crie uma aresta direcionada de y até x;
7          se lado esquerdo de x ⊂ lado direito de x então
8              Crie uma aresta de loop em x;
9      Seja B o conjunto de todas as operações binárias e ternárias do código suspeito;
10     para todo z ∈ B, faça
11         para todo w ∈ Vértices[], faça
12             se w ⊂ z então
13                 Crie uma aresta direcionada de w até z;
14             fimpara;
15         se lado esquerdo de z ⊂ lado direito de z então
16             Crie uma aresta de loop em z;
17     para toda operação Unária faça
18         Crie uma aresta de loop;
19 fim
```

usando as técnicas de ofuscação já discutidas neste trabalho antes da criação dos grafos de comportamento correspondentes.

O aspecto visual do grafo gerado em relação a versão do trabalho de [Kim and Moon 2010], mostra que o novo é mais sucinto e relaciona as operações unárias, binárias e ternárias ao invés das linhas de código. Isso simplifica uma validação manual, por exemplo. Outro aspecto considerável é o tamanho dos grafos. A Figura 8 mostra como o grafo de comportamento do algoritmo de fatorial ficaria usando o [Kim and Moon 2010] e usando a estratégia apresentada neste trabalho.

Nota-se que o tamanho do grafo, além da simplicidade na visualização, mostra uma redução considerável. Obviamente, as expressões nos vértices poderiam ser convertidas em números para simplificar, mas a decisão de deixá-los foi para permitir a avaliação do mecanismo de criação de dependências. Contudo, o uso de números deve ser usado em uma aplicação que use o mecanismo proposto, pois reduziria o consumo de memória criado pelos rótulos.

A garantia de continuidade de funcionamento das aplicações legadas é primordial. Por isso, a mudança aplicada por este trabalho não muda o grafo reduzido gerado após a fase de podas no grafo de comportamento. O processo de poda realizada no grafo de comportamento segue as regras de [Kim and Moon 2010].

- Elimine todos os vértices:
  - Com apenas uma aresta de saída sem nenhuma aresta de entrada;
  - Com somente uma aresta de entrada e nenhuma aresta de saída (significa que o vértice usa o valor de um anterior);
  - Com somente uma aresta de entrada e uma de saída (é um dado ou valor que está sendo “encaminhado” para outro);
  - Que não tenha nenhuma aresta entrando ou saindo (é um vértice considerado que não é necessário por ser redundante).

Para exemplificar, os exemplos da Seção 3 foram reduzidos de acordo com as regras descritas. A Figura 5 apresenta o resultado do grafo de comportamento da Figura 2. A Figura 6 o resultado do grafo de comportamento com reordenação de código e renomeação de variáveis da Figura 3. E, finalmente a Figura 7 o resultado da poda para o grafo de comportamento com lixo da Figura 4. Os grafos resultantes são iguais. Isso já era esperado, uma vez que todos representam a implementação do mesmo algoritmo de fatorial.

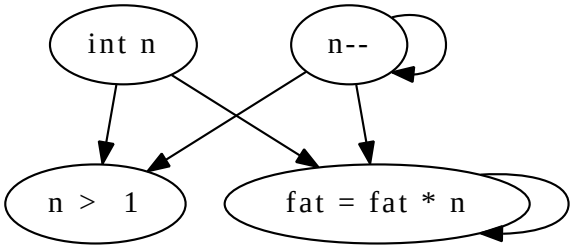

Figura 5. Grafo reduzido após a poda de [Kim and Moon 2010] no grafo da Figura 2.

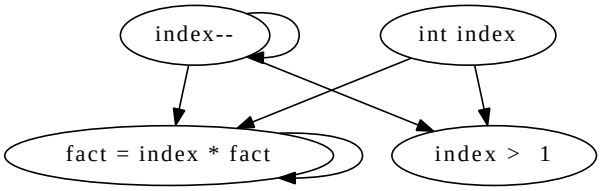

Figura 6. Grafo reduzido após a poda de [Kim and Moon 2010] no grafo de comportamento com renomeação e reordenação de variáveis da Figura 3.

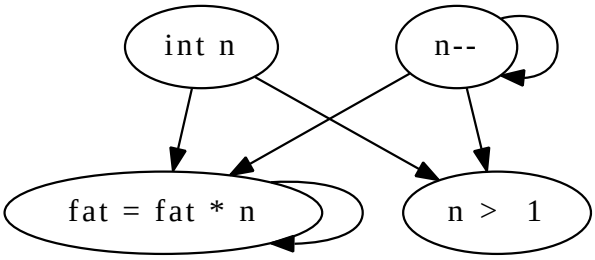

Figura 7. Grafo reduzido após a poda de [Kim and Moon 2010] no grafo de comportamento com inserção de lixo da Figura 4.

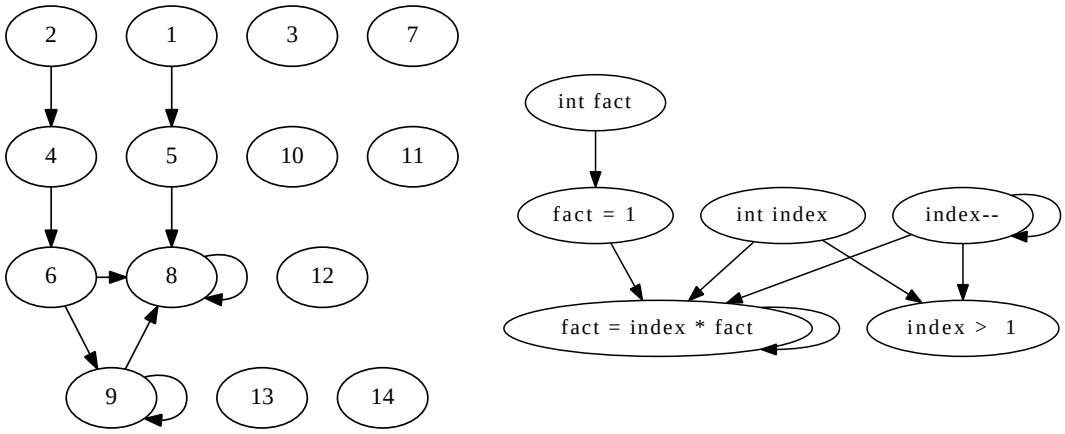

Figura 8. Grafo de comportamento gerado pela estratégia de [Kim and Moon 2010] e pelo Mapper.

A redução em relação é apresentada na Tabela 1. Essa tabela mostra a quantidade de vértices que cada estratégia cria quando um código com um determinado número de linhas é fornecido. A primeira coluna da tabela representa a quantidade de linhas de um código fonte em C. A segunda coluna representa a quantidade de vértices gerados por [Kim and Moon 2010]. A quantidade de vértices é superior a quantidade de linhas pois quando um código possui estruturas de controle de repetição, o número de linhas aumenta.

Por exemplo, quando um laço `for` é encontrado, o algoritmo de [Kim and Moon 2010] realiza a substituição do laço por uma entrada `if` e expande a expressão de controle do `for` dentro do corpo da nova declaração criada em conjunto com um mecanismo de `goto` para simular o laço. Por este motivo, o número de linhas cresce, e consequentemente o número de vértices também cresce. A terceira coluna apresenta a quantidade de vértices gerada por Mapper. E a última, a taxa de redução da quantidade de vértices do Mapper em relação ao [Kim and Moon 2010].

Tabela 1. Quantidade de vértices gerada por cada estratégia de mapeamento de informações no grafo de comportamento e a taxa de redução do seu tamanho.

Nº linhas	Kim	Mapper	Redução
10	16	6	62,5%
12	17	7	58,8%
17	22	13	41%
27	35	24	31,5%
38	38	34	21,8%
48	64	47	26,6%
60	95	66	30,5%
80	103	67	35%
100	132	80	39,4%

Nota-se, pelas informações da tabela, que a redução do tamanho do grafo é de, no mínimo, 20%. Além disso, o uso dos rótulos no Mapper foi mantido apenas para simplificar a

visualização. Em uma implementação prática, eles poderiam ser numerados para reduzir o uso de memória por vértice.

5. Conclusão

A técnica de geração de grafos de comportamento é utilizada pois os *malwares* podem mudar a sua estrutura interna sem modificar sua lógica semântica. Dessa forma, como apresentado no trabalho, mostra que a extração após a redução do grafo de comportamento mapeia corretamente o núcleo de significado de uma aplicação (suas estruturas mais importantes). Para construção destes grafos foram utilizadas bibliotecas NetworkX e PyCParser para apoio ao programa desenvolvido em Python.

Para determinar a eficiência do algoritmo criado em relação as técnicas aplicadas em trabalhos anteriores, foram avaliados códigos fonte de *malware* e suas diversas variantes. As variantes usaram técnicas clássicas de ofuscação como renomeação e reordenação de variáveis, inserção de lixo no código, alteração de fluxo, etc. A avaliação serve para coletar os grafos de comportamento de *malwares* e montar uma base de conhecimento. Para determinar se uma nova aplicação em análise é ou não um programa malicioso, basta criar o seu grafo de comportamento e compará-lo com a base conhecida. Se o grafo da aplicação foi similar a algum da base, com uma certa confiança é possível afirmar que a nova aplicação é uma variante ou não de um *malware* conhecido.

Os resultados alcançados mostram que o algoritmo foi capaz de reduzir o número de etapas necessárias para criar o grafo de comportamento, ou seja, não é necessário enumerar linhas, criar grafos de fluxo ou qualquer outro intermediário. Além disso, o grafo criado é menor e manteve as características semânticas. A taxa de redução do grafo alcançou valores próximos a 63% de redução do grafo total de entrada.

Embora não seja o objetivo, espera-se que a redução do grafo de entrada cause um impacto no tempo de execução de outros trabalhos que usam esse grafo como entrada. Além disso, também é esperada uma redução do uso de memória e o tempo de processamento do grafo para o grafo reduzido, acelerando os mecanismos atuais de detecção baseados em heurísticas para o problema de isomorfismo de subgrafos.

Como trabalho futuro será desenvolvida uma técnica para detecção de *malwares* polimórficos usando a estratégia de isomorfismo de subgrafos e um módulo para o antivírus livre Clamav. O objetivo do módulo é comparar as diferentes estratégias de detecção por comportamento em um ambiente não simulado. Além disso, alguns mecanismos como mutação e vírus com compactação de cabeçalho serão exploradas para aperfeiçoar a técnica de redução do grafo de comportamento.

Referências

[Bishop 2004] Bishop, M. (2004). *Introduction to Computer Security*. Addison-Wesley Professional, 1ª edition. ISBN-10: 0321247442, ISBN-13: 978-0321247445.

[Choi et al. 2014] Choi, H., Kim, J., and Moon, B.-R. (2014). A Hybrid Incremental Genetic Algorithm for Subgraph Isomorphism Problem. In *Proceedings of the 16th Conference on Genetic and Evolutionary Computation (GECCO'14)*, pages 445–452. ACM.

[Choi et al. 2012] Choi, J., Yoon, Y., and Moon, B.-R. (2012). An Efficient Genetic Algorithm for Subgraph Isomorphism. In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation ((GECCO'12))*, pages 361–368. ACM.

[Cozzolino et al. 2012] Cozzolino, M. F., Martins, G. B., Souto, E., and Deus, F. E. G. (2012). Detecção de variações de malware metamórfico por meio de normalização de código e identificação de subfluxos. In *XII Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais (SBSeg'12)*, pages 30–43, Curitiba, PR, BR.

[Grégio 2012] Grégio, A. R. A. (2012). *Malware Behavior*. PhD thesis, Universidade Estadual de Campinas (UNICAMP).

[Kim and Moon 2013] Kim, J. and Moon, B. (2013). Disguised Malware Script Detection System Using Hybrid Genetic Algorithm. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC'13)*, pages 182–187, Coimbra, Portugal.

[Kim and Moon 2010] Kim, K. and Moon, B. (2010). Malware Detection based on Dependency Graph using Hybrid Genetic Algorithm. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation (GECCO'10)*, pages 1211–1218, Portland, Oregon, USA.

[Kolbitsch et al. 2009] Kolbitsch, C., Comparetti, P. M., Kruegel, C., Kirda, E., Zhou, X., and Wang, X. (2009). Effective and Efficient Malware Detection at the End Host. In *Proceedings of the 18th Conference on USENIX Security Symposium (USENIX'09)*, pages 351–366, Berkley, CA, USA.

[Martins et al. 2014] Martins, G. B., Souto, E., Freitas, R., and Feitosa, E. (2014). Estruturas virtuais e diferenciação de vértices em grafos de dependência para detecção de malware metamórfico. In *XIV Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais (SBSeg'14)*, pages 260–273, Belo Horizonte, MG, BR.

[Network 2015] Network (2015). NetworkX: Software package for the creation, manipulation, and study of the structure. `https://networkx.github.io/`. [Online; acessado 10-Jul-2015].

[Prechelt et al. 2000] Prechelt, L., Malpohl, G., and Phlippsen, M. (2000). JPlag: Finding Plagiarisms Among a Set of Programs. Technical report.

[PyCParser 2015] PyCParser (2015). PyCParser: A complete parser of the C language. `https://pypi.python.org/pypi/pycparser`. [Online; acessado 10-Jul-2015].

[Report 2014] Report, M. (2014). The Economic Impact of Cybercrime and Cyber Espionage. `http://bit.ly/1fOFAQS`. [Online; acessado 10-Jul-2015].

[Schleimer 2003] Schleimer, S. (2003). Winnowing: Local Algorithms for Document Fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD'03)*, pages 76–85. ACM Press.

[UPX 2015] UPX (2015). UPX: the Ultimate Packer for eXecutables. `http://upx.sourceforge.net`. [Online; acessado 10-Jul-2015].