Bifocal Agent: identificando automaticamente funções maliciosas para aumentar o foco do analista de *malware*

Leonardo Gonçalves Chahud[1], **Rafael Oliveira da Rocha**[1], **Lourenço Alves Pereira Jr.**[1], **Idilio Drago**[4]

[1]Instituto Tecnológico de Aeronáutica — ITA

[2]Università di Torino, Italia — UNITO

{leonardo.chahud.101674,rafaelror,ljr}@ita.br

Abstract. *Although there are various solutions for automatic detection of malicious components on an analyzed executable, malware analysis is still predominantly a manual process, with the human analyst being its bottleneck. Recent works identify suspicious regions in code, reducing the analyst's effort. However, such solutions either are signature-based or generate many false positives. To overcome this challenge, we propose the Bifocal Agent, which operates at two distinct levels of granularity (function and basic block). The solution also uses new features to improve the detection of malicious functions. Experiments have shown that the solution increased the area under the ROC curve of the state-of-the-art related works by 17% and reduced false positives over a third.*

Resumo. *Embora existam diversas soluções de detecção automática de componentes maliciosos, a análise de malware ainda é um processo realizado predominantemente de forma manual, tendo como gargalo o analista humano. Trabalhos recentes foram capazes de identificar regiões suspeitas no código, reduzindo o esforço do analista. Entretanto, tais soluções são baseadas em assinaturas ou geram muitos falsos positivos. Para superar esse desafio, propomos o Bifocal Agent, que atua em dois níveis de granularidade distintos (função e bloco básico) e utiliza novas features para melhorar a detecção de funções maliciosas. Em experimentos, a solução aumentou em 17% a área sob a curva ROC do estado-da-arte e reduziu em mais de um terço os falsos positivos.*

1. Introdução

O cenário atual de cibersegurança é marcado por um grande número de tentativas de infecção por *malwares*. Somente no ano de 2023, as ferramentas de proteção da Kaspersky bloquearam mais de 430 milhões de ataques relacionados a *malwares* [Kaspersky 2023]. Evidentemente, isto representa apenas os ataques bloqueados, e não os ataques em sua totalidade, enfatizando a magnitude e atualidade do problema. Em meio a este cenário de infestação de arquivos maliciosos, profissionais de segurança precisam analisar amostras de *malware*, entendendo o que o artefato faz e como ele executa suas ações, de modo a identificar os índices de comprometimento (IoC) e as técnicas, táticas e procedimentos (TTP) do atacante. Esse processo quase sempre envolve a utilização da técnica de engenharia reversa, tendo o analista a laboriosa tarefa de interpretar o código e extrair dele informações relevantes [Yong Wong et al. 2021].

Neste contexto, diversos tipos de soluções já foram propostas para facilitar esse trabalho: geração automática de regras Yara [David and Netanyahu 2015, Coscia et al. 2023, Li et al. 2023]; identificação de traços e execução mais prováveis [Alrawi et al. 2021, Ruaro et al. 2022]; e análise do código [Zhong et al. 2013, Jones et al. 2016], estes últimos limitando-se a analisar o código como um todo, sem destacar trechos relevantes. Apesar de úteis, nenhuma das soluções apresentadas é capaz de dispensar o trabalho do analista humano, que possui uma rotina sobrecarregada de trabalho [Gutman 2019] e ainda precisa analisar uma grande quantidade de código para identificar comportamentos maliciosos.

Nesse sentido, o principal gargalo do analista de *malware* se dá sobre o estudo de seções de código inúteis as quais não acrescentam nenhum valor para o entendimento do programa e suas intenções. Isto ocorre porque, ao analisar funções que não são maliciosas, tempo considerável é gasto sem nenhum retorno significativo para a compreensão do binário. Desse modo, visando melhorar o processo de análise, novas soluções utilizando técnicas de aprendizado de máquina surgem, ainda que em escala pequena, como ferramentas auxiliares ao analista. A utilização destas técnicas vem sendo empregada por diversos trabalhos [Novkovic and Groš 2016, Downing et al. 2021], os quais demonstraram serem, de fato, beneficentes na rotina de trabalho do profissional da área. Ainda, a eficiência de tais técnicas foi demonstrada, sobretudo, na identificação de porções de código malicioso dentro de um executável.

Em vista disso, o presente trabalho propõe uma solução atuando na identificação de regiões maliciosas de código dada uma amostra de *malware* no contexto do sistema operacional *Windows*, avançando o estado-da-arte. Assim, o *Bifocal Agent* visa auxiliar o analista no principal ponto de atraso de sua linha de trabalho, através de uma nova metodologia utilizando a combinação de perspectivas de análise sob duas granularidades: blocos básicos e funções. São utilizados dois *autoencoders*, cada um treinado com o mesmo *dataset* de amostras benignas, porém com granularidades diferentes. Por fim, há a aplicação de uma heurística para determinar a natureza de uma função dentro do programa analisado.

As principais contribuições deste trabalho podem ser condensadas nos seguintes pontos: a) criação de um novo método de classificação de porções de código utilizando aprendizado de máquina não-supervisionado em conjunto com diferentes perspectivas de granularidade; b) criação de novos atributos mais eficientes para abstrair funcionalidades, sobretudo a nível de granularidade de funções; c) proposição de mudança de granularidade para funções ao invés de blocos básicos, alcançando melhor desempenho.

O restante deste artigo está estruturado da seguinte forma: na seção 2, são apresentados os trabalhos relacionados; a seção 3 consiste na arquitetura da solução proposta; na seção 4, a metodologia utilizada para a obtenção e avaliação dos resultados é explicada; em seguida, a seção 5 apresenta os resultados em conjunto com discussões; por fim, a seção 6 compreende as conclusões do trabalho.

2. Trabalhos Relacionados

Propostas voltadas à identificação de programas maliciosos são majoritárias no contexto de estudo de *malwares*. Contudo, estas soluções apresentam respostas à parte do problema: classificar a natureza de um programa executável como um todo, considerando-o

como bloco único de análise [Raff et al. 2017, David and Netanyahu 2015]; deixando de fora a identificação de funções maliciosas dentro do programa. Dessa maneira, na perspectiva do analista de *malware*, dado que este receba amostras benignas e malignas para análise, apenas o problema inicial de classificação do programa foi resolvido. Neste artigo, o foco se dá sobre a identificação de funções anômalas, definidas como funções que exercem comportamentos maliciosos como: comunicação com endereços IP de servidores de comando e controle; mecanismos de persistência; e evasão de sistemas de segurança.

Tabela 1. Tabela de funcionalidades presentes em cada solução.

Características	**DeepReflect**	**Jarv1s**	**CodeAnalyzer**	**BifocalAgent**
Aprendizado de Máquina	X		X	X
Blocos básicos	X			X
Funções		X	X	X
Atributos de API	X		X	X
Densidade API				X
Densidade de Leitura				X
Densidade de Escrita				X

Em [Downing et al. 2021], os autores propuseram uma solução, o *DeepReflect*, capaz de identificar regiões de interesse (RoIs) no *malware* analisado através da avaliação do erro de reconstrução obtido por um modelo *autoencoder* treinado sob amostras benignas. A solução estabelece um conjunto de atributos os quais representam os blocos básicos dentro do executável. Assim, o *autoencoder* é treinado sob blocos básicos benignos e, caso o erro de reconstrução de um bloco básico qualquer esteja acima de um limiar estabelecido, o bloco básico é considerado anômalo de modo que, consequentemente, a função em que o bloco está localizado também é considerada de interesse para a análise. Feita a identificação de funções as quais contém blocos anômalos, um novo vetor de atributos é criado para cada função de interesse, através de um cálculo o qual, por padrão, consiste na média dos atributos dos blocos básicos da função em questão. Em seguida, o algoritmo HDBSCAN é utilizado para realizar o processo de agrupamento de funções similares. Portanto, a medida que o analista atribui rótulos (e.g persistência, keylogger) para as funções, estes rótulos são propagados para funções pertencentes ao mesmo grupo. Entretanto, em relação à identificação de funções anômalas, os autores mostraram que a taxa de falso positivos é consideravelmente alta, com a solução proposta possuindo baixa precisão, sendo este um ponto indicado para ser melhorado em trabalhos futuros.

[Molloy et al. 2022] propuseram uma aplicação, denominada *Jarv1s*, capaz de decompor uma amostra de *malware* em unidades funcionais (funções em linguagem de montagem), e, a partir destas unidades, extrair informações, denominadas pelos autores como fenótipos, as quais serão utilizadas por um mecanismo de busca baseado em assinatura para associar as funcionalidades encontradas com famílias de *malware* anteriormente cadastradas. Os autores utilizam como granularidade funções em linguagem de montagem para a extração das características, contudo utilizam métodos baseados em assinatura argumentando maior rapidez e explicabilidade. É evidente, no entanto, que do ponto de vista do analista de *malware*, a solução proposta apenas tangencia o problema de identificação de códigos de interesse, focando em um processo de triagem, benéfico no

contexto de *Cyber Threat Intelligence*. Além disso, a utilização de técnicas baseadas em assinatura diminui a robustez do método, incapacitando a solução de detectar funcionalidades de amostras de *malware* novas ou arbitrariamente modificadas.

Já em [Zhong et al. 2013], os autores propuseram um modelo de identificação de variantes através da decomposição dos *malwares* em cadeias de comandos subsequentes mais longas. A solução apresentadada utiliza de *n*-gramas, similar ao que acontece em [Molloy et al. 2022], para representar cada instrução em linguagem de montagem, com a granularidade de trabalho também a nível de funções. Além disso, atributos numéricos relacionados à complexidade cíclica, número de instruções e número de argumentos de uma função são também considerados. Em seguida, o agrupamento funciona de maneira incremental, para evitar o custo de treinamento recorrente. Por fim, a solução proposta pelos autores, o *CodeAnalyzer*, alcança uma acurácia na classificação, métrica apontada como a mais importante para a mensuração de performance, de 61,6%. Desta maneira, apesar do objetivo distinto de identificação de variantes de *malware*, os autores em [Zhong et al. 2013] acabam por tangenciar, ao tentar identificar funções maliciosas dentro de um executável, o presente trabalho, atingindo, contudo, uma acurácia baixa.

Com isso, o *BifocalAgent* diferencia-se das soluções propostas ao atuar com múltiplas granularidades, considerando, portanto, tanto blocos básicos como funções, conjuntamente com um novo vetor de atributos proposto. O resultados experimentais deste artigo demonstram maior eficiência da solução quando avaliada sob as mesmas métricas dos autores em [Downing et al. 2021]. Em contraste com [Molloy et al. 2022], o *BifocalAgent* utiliza modelo de aprendizado não-supervisionado através de um *autoencoder*, e limita seu escopo de trabalho para a identificação de regiões de interesse em uma amostra de *malware*. Ainda, a quantidade de falsos positivos foi minorada significativamente, alcançando uma precisão de 96%. Por fim, explicita-se mais claramente as diferenças entre os demais trabalhos na Tabela 1.

3. Bifocal Agent

3.1. Arquitetura

A Figura 1 representa a linha de trabalho do *BifocalAgent*. Antes do início da execução, entretanto, assume-se que os melhores *thresholds* tenham sido escolhidos para ambos modelos de *autoencoders*. Estes limiares são obtidos apenas uma vez, na etapa de avaliação dos modelos. Logo, a solução que o analista terá em mãos já estará com os limiares previamente configurados, não sendo necessário nenhum ajuste nesse sentido. No início, ocorre a extração e o armazenamento dos atributos considerando as duas granularidades: blocos básicos (Tabela 4) e funções (Tabela 3). Em seguida, os dados são passados para os modelos treinados com as suas respectivas granularidades, obtendo os erros de reconstrução para todas as funções da amostra analisada. Por fim, a aplicação identifica quais funções apresentam um erro maior do que o limiar de comparação, para cada granularidade, e, no caso de ambos modelos acordarem sobre os erros estarem acima dos seus respectivos limites, as funções são classificadas como malignas.

De modo a formalizar o funcionamento dessa aplicação, denotamos: os dois modelos que utilizam as duas granularidades por M_1 e M_2, com M_1 e M_2 treinados, respectivamente, com blocos básicos e funções; e o espaço amostral Ω representando o conjunto de todas as funções do executável analisado. Assim, sejam L_1 e L_2 conjuntos de funções

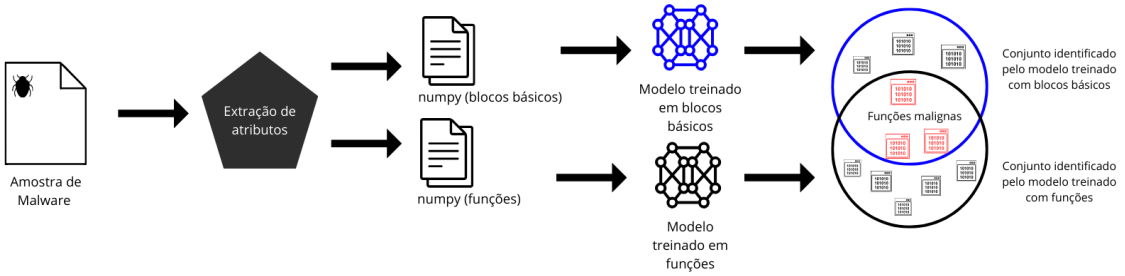


Figura 1. Arquitetura do *BifocalAgent*

identificadas como malignas pelos modelos M_1 e M_2, respectivamente, e, considerando a análise sobre o mesmo executável, sejam G_1 e G_2 conjuntos complementares de L_1 e L_2, nessa ordem, de modo que G_1 e G_2 consistem nas funções identificadas, por complementariedade, como benignas. Em seguida, denotamos por L_f e G_f os conjuntos que apresentam as funções malignas e benignas acordadas pelos dois modelos sendo, portanto, $L_f = L_1 \cap L_2$ e $G_f = G_1 \cap G_2$. Finalmente, estabeleceu-se o conceito de área cinza como sendo o conjunto de funções em que houve discordância acerca de suas naturezas pelos dois modelos, sendo estas funções encaminhadas ao analista para análise manual. Assim sendo, denota-se por D o conjunto que representa a área cinza e, portanto, da forma $D = (L_f \cup G_f)^c$.

3.2. Processo de Extração

O *BifocalAgent* representa as funções e os blocos básicos do binário analisado através dos vetores de atributos especificados nas Tabelas 3 e 4, respectivamente. Estes vetores são adotados considerando a granularidade utilizada: blocos básicos ou funções. A justificativa para essa decisão se deu através da constatação de que, experimentalmente, esta foi a metodologia que resultou em um melhor desempenho geral para a solução.

Além disso, o processo de extração assume que o executável de entrada não sofreu *packing* ou, caso tenha sofrido, este foi desempacotado corretamente. Esta suposição é de fundamental importância tendo em vista que a maioria dos atributos da Tabela 3 e 4 apresentam a contabilização de funções da *API* do *Windows*. Portanto, o nome destas funções deve ser acessível através do código em linguagem de montagem extraído, o que não pode ser garantido necessariamente caso o executável tenha sido empacotado. Deste modo, o processo de *unpacking*, estudado por outros trabalhos [Royal et al. 2006, Andriesse et al. 2017], se encontra fora do escopo deste artigo, havendo a possibilidade de exploração deste tópico em trabalhos futuros.

3.3. Erros de Reconstrução e Detecção de Regiões de Interesse

Um *autoencoder* é uma rede neural a qual apresenta uma compressão nas camadas intermediárias de neurônios, forçando a rede a aprender elementos fundamentais para a representação dos dados de entrada. Desse modo, erros de reconstrução são uma medida para a diferença entre os dados de entrada e os de saída, indicando numericamente o quão apta a rede se encontra para a reconstruir entradas de determinado tipo. Formalizando o cálculo dos erros de reconstrução, sejam os erros E_{f1} e E_{f2} calculados sob uma função F, obtidos pelos modelos treinados em funções e blocos básicos, respectivamente. Sejam também os componentes do *autoencoder*: codificador (E); decodificador (D). Seja X o

vetor de entrada com os atributos especificados na subseção anterior e $\hat{X}$ a entrada reconstruída, obtida na forma de $\hat{X} = D(E(X))$, em vista do *autoencoder*. Dessa maneira, para o modelo treinado com funções, tem-se que $E_{f1} = (X - \hat{X})^2$; enquanto para o modelo treinado em blocos básicos, o erro é calculado na forma de $E_{f2} = \frac{1}{m}\sum(x^{(i)} - \hat{x}^{(i)})^2$ com m sendo o número de blocos básicos pertencentes à F; $x^{(i)}$ a entrada do vetor de atributos correspondendo ao i-ésimo bloco básico; e $\hat{x}^{(i)}$ a entrada do vetor $x^{(i)}$ reconstruída pelo *autoencoder*. Em resumo, o erro de reconstrução de uma função, no caso do modelo treinado em blocos básicos, é a média dos erros quadráticos de reconstrução dos blocos básicos pertencentes àquela função.

Agora, após o processo de validação, suponha que os limiares ϕ_1 e ϕ_2 foram escolhidos pois foram considerados os melhores para os modelos treinados em funções e blocos básicos, respectivamente. Seja também R o conjunto de regiões de interesse identificadas, e os erros $e_{f1}^{(i)}$ e $e_{f2}^{(i)}$ calculados sob a i-ésima função $f^{(i)}$, considerando os modelos de funções e blocos básicos, nessa ordem. Desta maneira, tem-se que:

$$R = \left\{ f^{(i)} \mid e_{f1}^{(i)} \geq \phi_1 \text{ e } e_{f2}^{(i)} \geq \phi_2 \right\} \tag{1}$$

4. Metodologia

4.1. Dataset

O *dataset* [Lester 2021] utilizado por este trabalho consiste em um conjunto de arquivos executáveis na forma de *portable executables* contendo *drivers*, *dlls* e arquivos *exe*. O dataset possui 201,549 amostras de executáveis benignos e 86,812 amostras de *malwares*; estas últimas amostras referenciadas como amostras malignas ao longo deste trabalho. Este conjunto de dados também contém um arquivo *csv* com informações sobre os arquivos, incluindo rótulos para distinguir as amostras benignas das malignas. Assim, em um primeiro momento, foram separados os *malwares* dos demais arquivos e, em seguida, foram selecionadas amostras benignas de forma aleatória através da escolha de um prefixo qualquer no nome do arquivo. Por fim, foi observado que algumas amostras, apesar de suas naturezas benignas, passaram por um processo de *packing* de modo que foi necessário realizar um filtro para garantir a qualidade do *dataset* e a extração do vetor de atributos.

É de fundamental importância explicitar que este *dataset* não é o mesmo do utilizado por [Downing et al. 2021], dado que este último não foi disponibilizado pelos autores. Contudo, o conjunto de dados utilizado para avaliação, discutido em 4.4, é exatamente o mesmo. Portanto, pode-se comparar as métricas obtidas sob o conjunto de avaliação de modo a contrastar as soluções propostas.

4.1.1. Filtragem de binários

O processo de filtragem se deu de maneira automatizada através de um *script python* com a utilização da biblioteca *PyPackerDetect* que, por sua vez, utiliza de técnicas baseadas em assinaturas e heurísticas para a detecção de arquivos empacotados. Desse modo, caso

o *script* encontrasse qualquer suspeita ou irregularidade em um arquivo, este era descartado e deletado. As irregularidades consistem, na maioria das vezes, em nomes de seções incomuns, endereços de memória inconsistentes ou ausentes, e demais campos do cabeçalho *PE* modificados. Caso um *packer* tenha sido identificado por técnicas de assinatura, o arquivo também é descartado. Desse modo, foram selecionadas anteriormente cerca de 20,000 amostras benignas e, após o processo de filtro, restaram 13,868 amostras as quais foram utilizadas, posteriormente, para o treinamento das redes neurais.

4.2. Extração de Atributos

O processo de extração de *features* dos binários selecionados foi dividido em duas partes principais. A primeira consiste na extração de informações sobre as funções e blocos básicos de um executável através do *software* de análise e engenharia reversa *radare2*; seguido da estruturação destes dados extraídos em formato *json*, posteriormente salvos em formato compactado através da utilização das bibliotecas *zlib* e *msgpack*; ocorre, também nessa primeira etapa, um processo adicional de filtro para que funções geradas por compiladores ou pelo *debugger* do *radare2* não sejam consideradas. Já na segunda parte, ocorre a leitura destes arquivos compactados e a construção de vetores de atributos que serão posteriormente salvos em arquivos *numpy*. Importante ressaltar que o programa extrator permite a mudança de granularidade, sendo possível atribuir ao vetor de atributos correspondência direta entre funções ou blocos básicos do arquivo executável. Também é possível modificar o vetor de atributos, permitindo a realização de testes com combinações entre granularidades e vetores distintos.

4.2.1. Modificação de Atributos

O vetor de atributos do *DeepReflect* armazena a contagem simples de categorias de instruções *assembly* e funções da *API* do *Windows*, como mostrado na Tabela 2, e possui a seguinte estrutura: instruções de operações lógicas; instruções de *bit shifting*; instruções de movimentação de dados da pilha, registradores e portas; funções da *API* relacionadas a *dlls*, sistema de arquivos, rede, objetos, processos, registro do *Windows*, serviços, sincronização, informações do sistema e gerenciamento de tempo. Ele também armazena o *offspring* do bloco básico, isto é, quantas conexões saem desse bloco básico em direção a outros blocos. Por fim, é também utilizado o conceito de *betwenneess centrality*, ou intermediação, que consiste na mensuração de centralidade do bloco básico em uma rede direcionada formada por todos os blocos do executável.

Ao observar este vetor de atributos, não somente foi hipotetizada a mudança de granularidade para que a solução trabalhasse com funções inteiras, mas também, como consequência disto, a mudança na estrutura deste vetor (Tabela 3). Primeiramente, foram retiradas as métricas de intermediação e *offspring*, após ser demonstrado experimentalmente que não houve mudança significativa dos resultados com a remoção dessas características. Em seguida, foram adicionados os seguintes atributos, destacados em negrito na Tabela 3, todos referentes às funções da *API* do *Windows*: total de chamadas às funções; chamadas às funções de leitura; chamada às funções de escrita; densidade de chamadas de funções; densidade de leitura; densidade de escrita. Os três primeiros atributos são obtidos contabilizando o número de ocorrências de: todas as chamadas a *API*; de chamadas de leitura; e de chamadas de escrita, nessa ordem. Formalizando os três últimos atributos,

sejam D_a, D_r, D_w os atributos de densidade: de chamadas às funções da *API*; de chamadas de leitura e escrita, respectivamente; e I, A, R, W o número total de instruções de uma função ou bloco básico; o número total de chamadas a *API* do *Windows*; o número de chamadas a funções de leitura e escrita, nessa ordem, tem-se que:

$$D_a = \frac{A}{I} \qquad D_r = \frac{R}{A} \qquad D_w = \frac{W}{A} \tag{2}$$

Foi possível categorizar as funções da *API* do *Windows* entre leitura e escrita de forma semi-automática graças ao uso de verbos empregados na nomenclatura das funções. Exemplificando, funções com verbos *Get*, *Query*, *Read* naturalmente requisitam ou obtém informações do sistema sendo, portanto, classificadas como funções de leitura. Já funções contendo os verbos *Set*, *Update*, *Write* são classificadas como funções de escrita pois modificam objetos do sistema operacional como o sistema de arquivos, valores de chaves de registro e *sockets* de rede. Contudo, há funções disjuntas em relação as duas categorias, como as funções relacionadas à sincronização inter ou intra-processo. Neste caso, não houve nenhuma contabilização para os atributos de densidade.

Tabela 2. Atributos Originais do DeepReflect

Vetor DeepReflect		
arith_basic_math	trans_stack	api_dll
arith_logic_ops	trans_reg	api_file
arith_bit_shift	trans_port	api_network
api_object	api_process	api_registry
api_service	api_sync	api_sysinfo
api_time	betwenneess	offspring

Tabela 3. Atributos Utilizados Para Granularidade De Funções

Vetor Modificado 1		
arith_basic_math	trans_stack	api_dll
arith_logic_ops	trans_reg	api_file
arith_bit_shift	trans_port	api_network
api_object	api_process	api_registry
api_service	api_sync	api_sysinfo
api_time	**api_calls**	**api_read**
api_write	**api_density**	**api_read_density**
api_write_density	**instruction-count**	

Tabela 4. Atributos Utilizados Para Granularidade De Blocos Básicos

Vetor Modificado 2

arith_basic_math	trans_stack	api_dll
arith_logic_ops	trans_reg	api_file
arith_bit_shift	trans_port	api_network
api_object	api_process	api_registry
api_service	api_sync	api_sysinfo
api_time	betwenneess	offspring
api_calls	**api_read**	**api_write**
api_density	**api_read_density**	**api_write_density**
instruction-count		

4.2.2. Modificação de Granularidade

4.3. Pré-processamento

Uma vez extraídos os vetores de atributos de todos os executáveis do *dataset*, foi necessário realizar um tratamento dos dados para posterior utilização no treinamento dos *autoencoders*. Diversos *pipelines* de pré-processamento foram testados com as duas granularidades disponíveis (funções e blocos básicos), sendo os melhores *pipelines* apresentados nas subseções seguintes. O primeiro *pipeline* foi utilizado com granularidade a nível de funções, enquanto o segundo a nível de blocos básicos. Para mais, vale ressaltar que não houve nenhum pré-processamento no trabalho original do *DeepReflect*, senão a normalização pelo valor máximo.

4.3.1. Características dos dados

Ao analisar os dados, em um primeiro momento, percebeu-se que o valor máximo da variável contendo o número total de instruções do bloco básico ou função estava na magnitude de 5 ordens de grandeza. Verificou-se, posteriormente, que o *radare2*, para alguns executáveis, provavelmente devido a algum erro, analisava todo o executável como uma função ou um bloco básico apenas, desbalanceando o *dataset*. Em um segundo momento de análise, observou-se, ao traçar histogramas para as variáveis, uma forte tendência à assimetria positiva em grande parte dos atributos, além de escalas diferentes. Assim, houve, por fim, a constatação de que a maioria dos vetores de atributos não possuíam chamadas a funções da *API* do *Windows*, numa escala de desbalanceamento de 95% e 5%.

4.3.2. 1° Pipeline de pré-processamento

Neste primeiro *pipeline*, trabalhando com granularidade a nível de função, foi feito, primeiramente, a exclusão de vetores praticamente vazios que apresentavam somente o atributo *instruction-count* preenchido. Em seguida, foram selecionados os vetores, levando em conta ainda este mesmo atributo, até o quantil 95% para que as amostras discrepantes fossem excluídas. Logo após, foi criado um atributo booliano temporário que indicava a presença de chamadas a *APIs* do *Windows* no vetor. Com isso, foi possível realizar uma amostragem para o conjunto de treino e teste de modo que vetores os quais apresentassem chamadas a estas funções estivessem em quantidades compatíveis proporcionalmente nos

dois conjuntos de dados. Em um último momento, os atributos foram normalizados simplesmente realizando a divisão pelo valor máximo, assim como é feito pelo *DeepReflect*.

4.3.3. 2° Pipeline de pré-processamento

Trabalhando com granularidade a nível de blocos básicos, este segundo *pipeline* inicia o pré processamento realizando a limpeza dos vetores praticamente vazios, como explicado no primeiro *pipeline*. Em seguida, para tratar valores extremos, sabendo da forte assimetria positiva no atributo *instruction-count*, utilizou-se dos limites de Tukey com fator de escala 1,5 para filtrar o conjunto de dados. Após esta etapa, foi criado um atributo temporário consistindo na entropia calculada sob cada vetor para que, em seguida, fosse utilizado novamente os limites de Tukey com o mesmo fator de escala (1,5), porém excluindo somente os vetores os quais apresentavam entropia abaixo do limite inferior. Agora, dado o desbalanceamento em relação às chamadas a *API*, foi utilizado, devido ao tamanho substancial do conjunto de dados, a técnica de subamostragem aleatória, balanceando o conjunto igualmente entre vetores que possuem chamadas e os que não possuem. Ao fim do processo, houve a normalização realizada através da divisão pelo valor máximo.

4.4. Avaliação da Solução e Obtenção de Thresholds

A avaliação é feita sob um conjunto de binários maliciosos, disponibilizados pelos autores em [Downing et al. 2021], pertencentes às famílias de *malware rbot*, *pegasus* e *carbanak*. Cada um dos executáveis pertencentes a uma dessas famílias possui um arquivo, denominado arquivo de anotação, o qual contém rótulos de algumas funções presentes no programa. Estes rótulos são utilizados em conjunto com os erros de reconstrução obtidos nas funções para a criação das curvas ROC. Dessa maneira, primeiro são traçadas as curvas para cada família de *malware* individualmente através da combinação, caso haja mais de um executável em determinada família, dos rótulos e dos erros de reconstrução de executáveis que pertencem àquela família. Ao final do processo, tem-se três curvas ROC, cada uma representando uma família de modo que, com isto, a avaliação é finalizada traçando uma última curva combinando os resultados destas outras três.

Aprofundando a discussão acerca da obtenção das curvas e ROC e, consequentemente, dos *thresholds*, pode-se afirmar que são necessários dois artefatos para traçar as curvas: rótulos das funções (maligna ou benigna); *scores* que, neste caso, são os erros de reconstrução obtidos. Os rótulos são obtidos pelo conjunto de avaliação oferecido pelos autores em [Downing et al. 2021], enquanto os *scores* são obtidos utilizando o *autoencoder* já treinado. Sabe-se ainda da ocorrência de um *trade-off* nas taxas de falsos positivos e verdadeiros positivos, permitindo selecionar um *threshold* em uma taxa de verdadeiros positivos específica. Dessa maneira, para a avaliação da solução que utiliza ambas granularidades, foram utilizados *thresholds*, para cada granularidade, onde a taxa de verdadeiros positivos é 80%.

O método para avaliação discutido previamente não se difere sob nenhum aspecto do utilizado em [Downing et al. 2021], para que a comparação se faça justa. Dessa forma, os mesmos binários das famílias de *malware* citadas, bem como os mesmos arquivos de anotação são utilizados, sem nenhuma modificação. Ocorre que devido ao fato de em [Downing et al. 2021] o programa de extração utilizado (*Binary Ninja*) ser diferente,

Tabela 5. Área sobre a curva das curvas ROC.

Solução	Dataset	Granularidade	Rbot	Pegasus	Carbanak	Combinado
BifocalAgent	BifocalAgent	Função	0.7242	0.7677	0.8423	0.8886
BifocalAgent	BifocalAgent	Bloco básico	0.8385	0.6109	0.6756	0.7989
DeepReflect	BifocalAgent	Função	0.6359	0.6459	0.8027	0.7120
DeepReflect	BifocalAgent	Bloco Básico	0.6818	0.7349	0.8314	0.7612
DeepReflect	DeepReflect	Bloco Básico	0.8429	0.7926	0.7634	0.8319

algumas funções, com o mesmo endereço de memória, apresentam tamanhos distintos e, portanto, vetores de atributos correspondentes diferentes, adicionando certa variabilidade no cálculo dos resultados. Contudo, desconsideradas as diferenças acerca da execução do processo de avaliação, a metodologia se manteve a mesma, sendo passível de comparação.

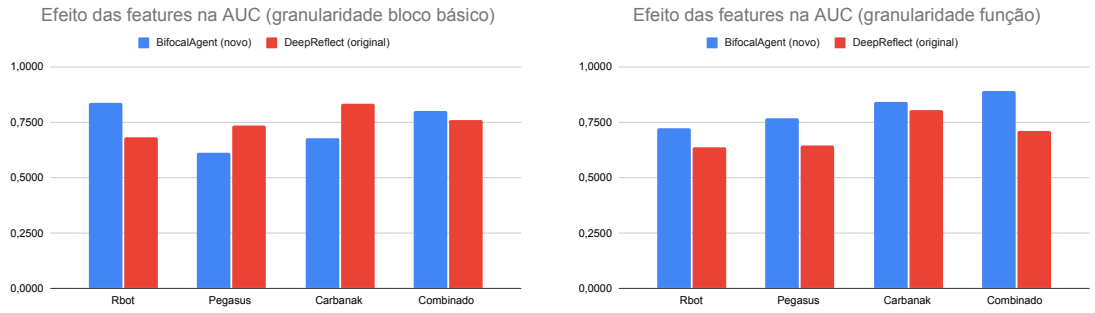
5. Resultados e Discussões

5.1. Impacto das modificações na performance

A Tabela 6 apresenta os resultados de performance (área sobre a curva ROC) obtidos em todos os cenários experimentados (o que inclui o *DeepReflect* treinado com as mesmas amostras benignas utilizadas no *BifocalAgent*), além do resultado divulgado no artigo [Downing et al. 2021], cujo modelo foi treinado em outro *dataset*, este último indisponível para a realização de testes. Nas subseções a seguir, cada cenário será avaliado separadamente de maneira mais detalhada.

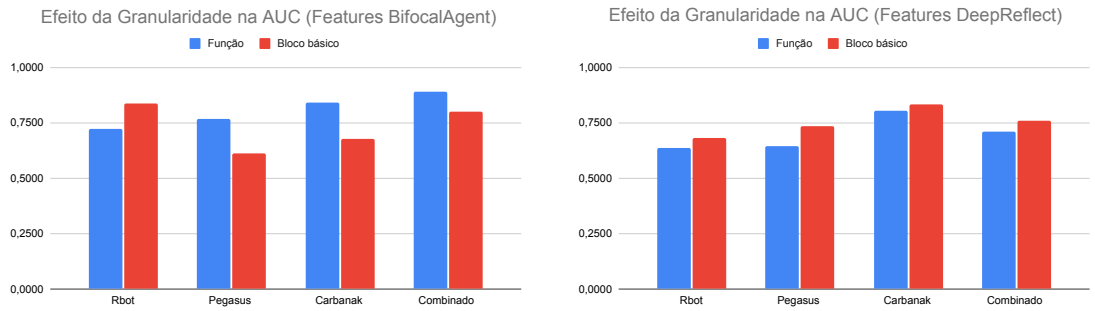
Impacto da engenharia de atributos no desempenho: conforme pode ser observado na Figura 2(a), a engenharia de atributos teve um ligeiro impacto positivo no desempenho geral do modelo, apresentando melhoria na família *rbot*, e piora nas famílias *carbanak* e *pegasus*, considerando granularidade a nível de blocos básicos. Contudo, ao considerar granularidade de função, na Figura 2(b), observa-se melhora geral de aproximadamente 25% se comparado ao *DeepReflect*, com desempenho superior em todas as famílias de *malware* consideradas. Estes resultados podem ser observados mais precisamente na Tabela 5, ao comparar a segunda e quarta linhas (mudanças a nível de blocos básicos), em conjunto com a primeira e terceira linhas (mudanças a nível de funções). Dessa forma, através de um vetor de atributos mais simplificado, foi obtido um resultado páreo considerando granularidade a nível de blocos básicos, porém superior quando se atua a nível de funções.

Impacto da granularidade no desempenho: Ao observar a Figura 3(a), nota-se que, para o novo conjunto de atributos, a alteração para a granularidade de funções gerou um impacto positivo no desempenho geral do modelo, aumentando a área sobre a curva ROC em 11%. Mais especificamente, houve uma queda de performance na família *rbot* de aproximadamente 16%, porém com um aumento no desempenho em 25% e 24% para as famílias *pegasus* e *carbanak*, respectivamente. Estes dados podem ser deduzidos ao examinar, na Tabela 5, as primeira e segunda linhas, comparando os dados das áreas sobre



(a) Granularidade no nível bloco básico (b) Granularidade no nível função.

Figura 2. Efeito da modificação do conjunto de *features* na AUC



(a) Conjunto novo de features. (b) Conjunto original de features.

Figura 3. Efeito da modificação da granularidade na AUC

as curvas individualmente. Ainda, conforme pode ser observado na Figura 3(b), ao se utilizar o conjunto original de *features* acompanhado desta granularidade, o desempenho do modelo sofre uma pequena redução de performance em todos os casos. Examinando novamente a Tabela 5, as reduções consistem em 13% para o *rbot*, 18% para o *pegasus*, e 5% para o *carbanak*.

Impacto das modificações em conjunto no desempenho: Na Tabela 5, é possível realizar a comparação das modificações propostas em relação a granularidade e modelo do vetor de atributos com dados originais apresentados em [Downing et al. 2021] (primeira e quinta linhas), bem como medir as diferenças entre a metodologia do *DeepReflect* assumindo o mesmo conjunto de dados para treinamento (primeira e quarta linhas). Nesse sentido, efetuando uma comparação mais detalhada tendo em vista o mesmo *dataset*, tem-se melhoras de 6%, 4% e 1% nas famílias *rbot*, *pegasus* e *carbanak*, respectivamente, com aumento da área combinada em aproximadamente 17%. Pode também ser observado uma AUC em torno de 7% maior do que o valor divulgado no artigo original.

5.2. Efetividade da solução nova

Para verificar a efetividade da solução proposta, foram selecionados dois *thresholds* para os modelos M_1 (treinado em funções) e M_2 (treinado sob blocos básicos), sendo estes os que apresentaram a melhor performance para a família *rbot*. Esta decisão se deu devido ao

fato de que o *rbot* é a família com mais funções classificadas manualmente pelos analistas e, portanto, torna-se uma opção sólida para deduzir os resultados. A família *rbot* conta com 92 funções malignas e 20 benignas, e foi testada nos três cenários: classificação pelo modelo M_1, apenas; pelo modelo M_2, somente; e pela combinação dos dois modelos (M_{12}), já exemplificada na seção que descreve a arquitetura da solução.

Observando os resultados apresentados na Tabela 6, pode-se constatar o seguinte:

- Melhora na precisão com a metodologia de granularidade múltipla, sendo o ganho em relação as funções e blocos básicos, respectivamente, de 4,5% e 9,7%;
- Diminuição proporcionalmente maior da quantidade de erros em relação a quantidade de acertos quando aplicada a metodologia de granularidade múltipla, explicitando um *trade-off* positivo;
- Diminuição, já antevista devido a conceitualização da área cinza, do *recall* ao considerar a granularidade múltipla. A diminuição do F1-Score é, por conseguinte, também esperada devido ao cálculo da média harmônica empregado; e
- Região cinzenta representando 35% do total de funções. Essa região pode ser priorizada para análise nos eventuais casos em que as funções classificadas como maliciosas não sejam suficientemente numerosas para a análise do código.

Tabela 6. Métricas de Performance

Granularidade	Funções	Blocos Básicos	Granularidade Múltipla
Precision	0.926	0.882	0.968
Recall	0.815	0.815	0.663
F1-Score	0.867	0.847	0.787
Acertos	89	85	67
Erros	23	27	5
Amostras Cinzas	0	0	40

5.3. Discussão e Limitações

Discussão sobre o impacto da modificações: em vista dos resultados apresentados anteriormente, a mudança do vetor de *features* com a inclusão de atributos de densidade de *API* e a utilização de granularidade no nível função trouxeram impacto positivo para a área sobre a curva ROC, o que indica melhores resultados tanto em relação aos falsos positivos quanto em relação aos falsos negativos quando comparados ao conjunto de *features* e granularidade originais. Entretanto, cabe ressaltar que os resultados variam bastante de família para família. Por exemplo: ao se utilizar as *features* novas, a granularidade bloco básico apresenta o melhor resultado para a família *rbot*, embora o resultado seja diverso com as demais famílias e no resultado consolidado. Essa característica indica que a combinação das duas granularidades pode gerar resultados mais precisos, ideia que impulsionou a criação do *BifocalAgent*, o qual obteve resultados os quais confirmaram a hipótese inicial de resultados com maior precisão. Conforme os [Downing et al. 2021] citaram em seu trabalho original, a qualidade do modelo é bastante afetada pelo conjunto de amostras benignas utilizada para o treino: quanto mais representativo for esse conjunto, melhor o modelo aprenderá o comportamento benigno. Por outro lado, conforme constatado nos experimentos, a capacidade de classificação varia de acordo com

a família. Nesse sentido, estudos mais profundos com um maior número de famílias de *malware* são necessários para compreender melhor as causas de erro de classificação e propor melhorias no treinamento.

Discussão sobre a efetividade da solução: o *BifocalAgent*, combinando as perspectivas de blocos básicos e funções em uma única aplicação, demonstrou-se bastante eficiente na redução dos erros cometidos, mantendo uma alta taxa de acerto. O aumento da precisão e a redução dos falsos positivos representam melhora significativa na rotina do analista de *malware*, visto que as funções encaminhadas para análise manual possuem grandes chances de serem, de fato, maliciosas. Enquanto um argumento pode ser feito afirmando que a criação da área cinza representa uma carga de trabalho adicional ao analista, este pode ser rebatido em vista da natureza incremental do processo de análise de modo que, conforme o analista avança e entende as principais ações do *malware*, através do estudo de funções maliciosas, as demais funcionalidades passam a ser antevistas, acelerando o restante da análise. Cabe ainda ressaltar que a abordagem utilizada para combinar as perspectivas de granularidade foi bastante simples, resumindo-se a considerar uma função maliciosa quando ambos os modelos (o de função e o de bloco básico) considerarem-na como tal. Entendemos que estudos mais profundos podem ser conduzidos no sentido de identificar melhores técnicas para consenso dos modelos, de modo a proporcionar ainda melhores resultados acerca da precisão.

Demais limitações: ataques adversariais podem ser empregados por atacantes no sentido de fazer com que o modelo classifique determinadas funções maliciosas sempre como benignas. Para mais, técnicas de ofuscação de código e *packing* podem ser utilizadas no sentido de prejudicar o processo de extração de atributos da amostra analisada, dado que parte dos atributos necessita da contabilização de chamadas de funções da *API* do *Windows* .Em relação ao *dataset* de avaliação, foi utilizado o mesmo dataset de [Downing et al. 2021] pelo fato de não termos conhecimento de outro conjunto de dados compatível com o propósito do trabalho. Entretanto, cabe ressaltar que esse *dataset*, além de apresentar somente três famílias de *malware*, é substancialmente desbalanceado tanto em relação ao número de funções benignas/maliciosas quanto em relação ao número de amostras das famílias. Entendemos que o primeiro tipo de desbalanceamento é positivo, pois é uma boa representação da realidade. Entretanto, o segundo desbalanceamento pode gerar distorções que atrapalham a avaliação dos resultado. Entendemos, portanto, que trabalhos futuros devem ser feitos no sentido de se criar um *dataset* com uma maior quantidade de famílias de *malware*, com grupos de famílias com quantidade de funções similares.

6. Conclusões

Neste trabalho avaliamos formas de melhorar a precisão de modelos de identificação de regiões suspeitas em um código, de modo a facilitar a pesada tarefa do analista de *malware*. Foram realizados testes com um novo conjunto de *features* e com uma granularidade maior (função, em vez de blocos básicos). As modificações propostas foram capazes de melhorar o desempenho do estado-da-arte em 17%, considerando a curva ROC como métrica de avaliação. Para reduzir o número de falsos positivos, propusemos ainda

uma nova abordagem para classificação da uma função utilizando o consenso de modelos que atuam em níveis de granularidade distintos. Essa nova abordagem foi capaz de melhorar a precisão da classificação e ainda de criar uma ”região cinzenta”, que pode ser utilizada para priorizar as funções a serem avaliadas nos casos em que as regiões classificadas como suspeitas não forem suficientes para a análise completa do código.

A condução de trabalhos futuros pode se dar sobre os seguintes pontos: (a) Criar um *dataset* com funções rotuladas que seja balanceado adequadamente e mais representativo. (b) melhorar a engenharia de atributos, dada a natureza flexível deste componente; (c) melhorar o pré-processamento dos conjuntos de treino e teste com algoritmos de sub-amostragem não aleatórios; (d) estudar formas mais eficientes para gerar o consenso dos modelos que atuam em granularidades distintas; (e) avaliar a solução sobre executáveis que passaram por processo de *packing*; (f) como complemento, aplicar os mesmos princípios em sistemas operacionais diferentes e avaliar os resultados obtidos com soluções já existentes; (g) avaliar comparativamente a solução proposta por [Downing et al. 2021] em relação à apresentada neste artigo, no caso de utilização de técnicas de *obfuscation* e *tampering* nos executáveis; (h) avaliar a performance da aplicação, em relação aos *thresholds* escolhidos, em outros *datasets* e executáveis.

Agradecimentos

Este trabalho tem apoio financeiro do Programa de Pós-graduação em Aplicações Operacionais—PPGAO/ITA, da Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP) processo #2020/09850-0 and #2022/00741-0, da CAPES e do CNPq.

Referências

Alrawi, O., Ike, M., Pruett, M., Kasturi, R. P., Barua, S., Hirani, T., Hill, B., and Saltaformaggio, B. (2021). Forecasting malware capabilities from cyber attack memory images. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3523–3540. USENIX Association.

Andriesse, D., Slowinska, A., and Bos, H. (2017). Compiler-agnostic function detection in binaries. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 177–189.

Coscia, A., Dentamaro, V., Galantucci, S., Maci, A., and Pirlo, G. (2023). Yamme: a yara-byte-signatures metamorphic mutation engine. *IEEE Transactions on Information Forensics and Security*, 18:4530–4545.

David, O. E. and Netanyahu, N. S. (2015). Deepsign: Deep learning for automatic malware signature generation and classification. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8.

Downing, E., Mirsky, Y., Park, K., and Lee, W. (2021). DeepReflect: Discovering malicious functionality through binary reconstruction. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3469–3486. USENIX Association.

Gutman, Y. (2019). Stop the churn, avoid burnout: How to keep your cybersecurity personnel. `https://assets.sentinelone.com/ciso/sentinel-one-stop-th`. Accessed: 2024-09-30.

Jones, L., Sellers, A., and Carlisle, M. (2016). Cardinal: similarity analysis to defeat malware compiler variations. In *2016 11th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 1–8.

Kaspersky (2023). Kaspersky Security Bulletin 2022. Statistics — securelist.com. `https://securelist.com/ksb-2023-statistics/111156/`. [Acessado em 20-Maio-2024].

Lester, M. (2021). Pe malware machine learning dataset. `https://practicalsecurityanalytics.com/pe-malware-machine-learning-dataset/`. Accessed: 2024-09-30.

Li, S., Ming, J., Qiu, P., Chen, Q., Liu, L., Bao, H., Wang, Q., and Jia, C. (2023). Packgenome: Automatically generating robust yara rules for accurate malware packer detection. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 3078–3092, New York, NY, USA. Association for Computing Machinery.

Molloy, C., Charland, P., Ding, S. H. H., and Fung, B. C. M. (2022). Jarv1s: Phenotype clone search for rapid zero-day malware triage and functional decomposition for cyber threat intelligence. In *2022 14th International Conference on Cyber Conflict: Keep Moving! (CyCon)*, volume 700, pages 385–403.

Novkovic, I. and Groš, S. (2016). Can malware analysts be assisted in their work using techniques from machine learning? In *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1408–1413.

Raff, E., Barker, J., Sylvester, J., Brandon, R., Catanzaro, B., and Nicholas, C. (2017). Malware detection by eating a whole exe.

Royal, P., Halpin, M., Dagon, D., Edmonds, R., and Lee, W. (2006). Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 289–300.

Ruaro, N., Pagani, F., Ortolani, S., Kruegel, C., and Vigna, G. (2022). Symbexcel: Automated analysis and understanding of malicious excel 4.0 macros. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1066–1081.

Yong Wong, M., Landen, M., Antonakakis, M., Blough, D. M., Redmiles, E. M., and Ahamad, M. (2021). An inside look into the practice of malware analysis. CCS '21, page 3053–3069, New York, NY, USA. Association for Computing Machinery.

Zhong, Y., Yamaki, H., Yamaguchi, Y., and Takakura, H. (2013). Ariguma code analyzer: Efficient variant detection by identifying common instruction sequences in malware families. In *2013 IEEE 37th Annual Computer Software and Applications Conference*, pages 11–20.