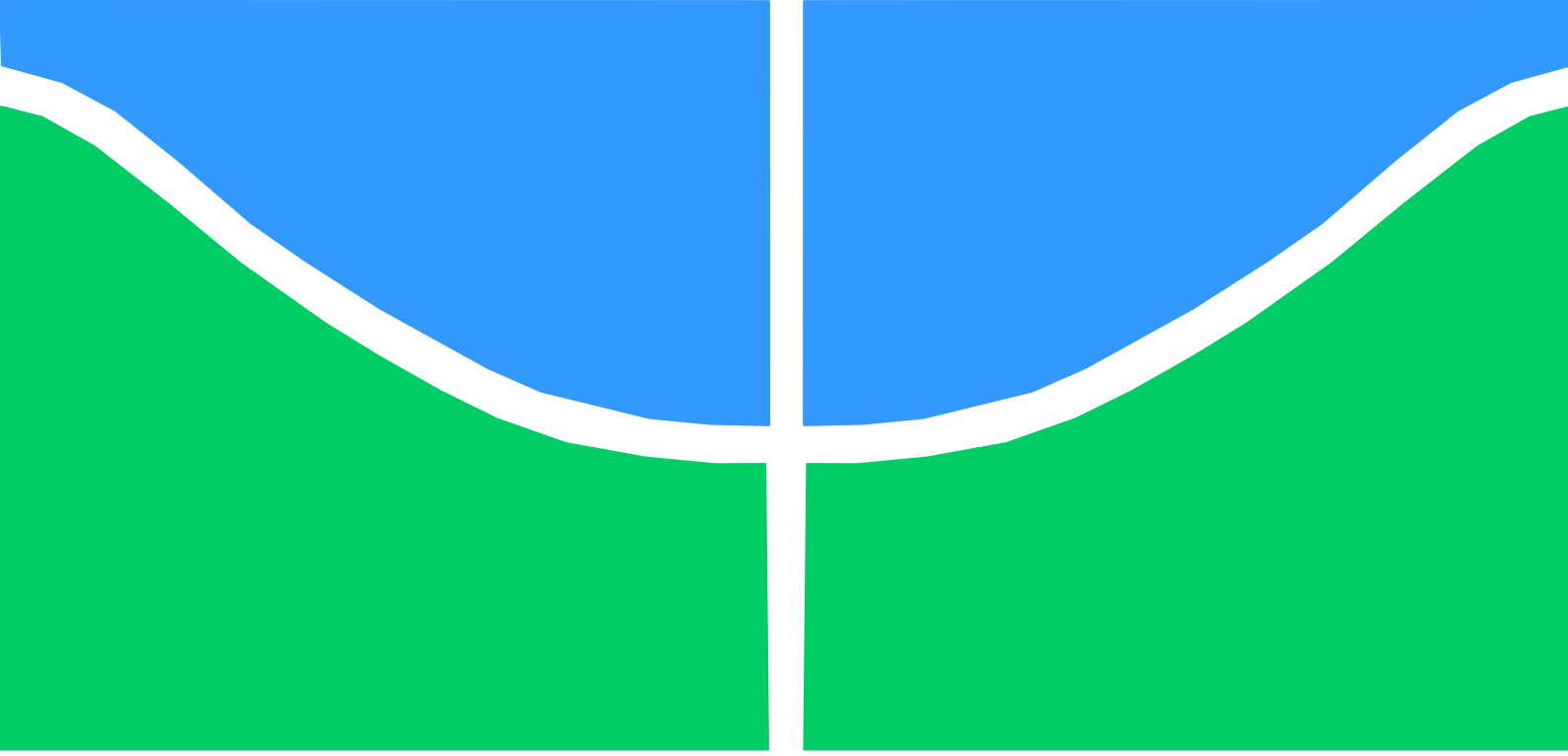

TRABALHO DE GRADUAÇÃO

TÉCNICAS DE DETECÇÃO E CLASSIFICAÇÃO DE MALWARES BASEADA NA VISUALIZAÇÃO DE BINÁRIOS

Johan Matos Coelho da Silva
Philipe Matos Coelho da Silva

Brasília, agosto de 2018

UNIVERSIDADE DE BRASÍLIA
Faculdade de Tecnologia

DEPARTAMENTO DE ENGENHARIA ELÉTRICA
UNIVERSIDADE DE BRASÍLIA
Faculdade de Tecnologia
Departamento de Engenharia Elétrica

TRABALHO DE GRADUAÇÃO

TÉCNICAS DE DETECÇÃO E CLASSIFICAÇÃO DE MALWARES BASEADA NA VISUALIZAÇÃO DE BINÁRIOS

Johan Matos Coelho da Silva
Philipe Matos Coelho da Silva

Trabalho de Conclusão de Curso submetido ao Departamento de Engenharia Elétrica, como parte dos requisitos necessários para obtenção do grau de Engenheiro de Redes de Comunicação

Banca Examinadora

Prof. Flávio Elias Gomes de Deus, Dr., UnB/ENE (Orientador)

Prof. Robson de Oliveira Albuquerque, Dr., UnB/ENE (Coorientador)

Prof. Georges Daniel Amvame Nze, Dr., UnB/ENE (Examinador)

"Sic Parvis Magna."[1]

(Sir Francis Drake)

[1] Grandeza a partir de pequenos começos.

Dedico este trabalho aos meus pais, que sempre se preocuparam em prover uma boa educação e criação.

Johan Matos Coelho da Silva

Dedico este trabalho aos meus pais, Carlos e Romira, que nunca pouparam esforços para a minha educação.

Philipe Matos Coelho da Silva

AGRADECIMENTOS

À minha família, pelo suporte, incentivo e incontáveis conselhos que me foram dados.
Aos meus amigos, pela compreensão e apoio.
Aos amigos de faculdade, que tornaram essa jornada mais agradável.
Aos professores que contribuíram para essa graduação e projeto, com ensinamentos, dicas e a orientação em temas desafiadores.

Johan Matos Coelho da Silva

À minha família, principalmente aos meus pais pelos ensinamentos e todo o suporte que me foram dados para que eu pudesse me dedicar aos estudos.
Aos meus amigos pelo companheirismo e torcida constante.
Aos professores orientadores deste trabalho, pelo norteamento e valiosas sugestões.
Agradeço aos bons professores que contribuíram para que tivesse uma boa formação acadêmica.
Aos técnicos e servidores que me propiciaram um ambiente de aprendizagem.

Philipe Matos Coelho da Silva

Resumo

O presente trabalho discorre sobre o cenário atual da segurança da informação, as ameaças constantes por parte dos *malwares* e as possibilidades inovadoras que permitam facilitar e agilizar a análise destes códigos maliciosos. A cada dia, milhares de novos tipos de *malwares* inundam a internet. Estes novos programas mal-intencionados não são de imediato reconhecidos por programas antivírus, pois essas ameaças precisam ser analisadas e então suas assinaturas são submetidas aos bancos de dados das companhias Antivírus. Por isso, aqui são apresentadas técnicas que auxiliam a identificação e classificação dessas ameaças com base em representações visuais de seus arquivos binários, permitindo lidar de forma mais ágil e eficiente com esse constante e incessante fluxo de novas ameaças. São apresentados algoritmos para a comparação de imagens e de *hashing* perceptivo, os quais serão aplicados às assinaturas visuais dos *malwares*. Em seguida, os mesmos serão comparados quanto à eficácia na classificação e deteção de *malwares*. Os resultados obtidos, estão dispostos no capítulo 4. Os métodos propostos alcançaram 94,2% de precisão na classificação em tipo de ameaça para um banco de dados de 2.134 amostras, 87,6% de acerto na detecção de 41 *malwares* que não foram detectados pelo antivírus escolhido como padrão para o trabalho e 100% de acerto para arquivos benignos. Ajustando-se os algoritmos aplicados, foi possível obter 100% na detecção destes 41 *malwares* com 81,5% de acerto em relação aos arquivos limpos.

Palavras-chave: *malwares*, *hashing*, visualização de dados, Segurança da Informação, MSE, SSIM.

Abstract

This paper discusses the current scenario of information security, malware threats and innovative possibilities for speeding up the analysis of these malicious codes. Every day, thousands of new types of malware flood on the internet. These new malicious codes are not immediately recognized by antivirus programs. Those malicious samples have to be analyzed, detected as harmful and then, submitted to the databases of antivirus companies. Therefore, some techniques are hereby presented aiming to assist the identification of malwares through their visual signature representation, allowing a different approach when dealing with this constant and never ceasing stream of new threats. Algorithms are presented for images and perceptual hashes comparison, which will be applied to the visual signatures of the malware that were obtained after applying data visualization techniques on the binaries of malicious programs. At the end, the effectiveness of the proposed algorithms is going to be evaluated, measuring how effective are malware classifications and detections with proposed methods. Results are shown in chapter 4. It's been achieved a 94,2% correct classification on malwares types using 2.134 samples and 87,6% detection rate on 41 malicious codes that weren't detected by the default antivirus program chosen while not detecting 100% of clean Windows files. Adjusting the algorithms, it was possible to achieve a 100% detection rate for those 41 samples with 81,5% of success on the clean files.

Keywords: malwares, hashing, Data Visualization, Information Security, MSE, SSIM.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS FONTE

LISTA DE TERMOS E SIGLAS

API	*Application Programming Interface*
AV	AntiVírus
DCT	*Discrete Cosine Transform*
DLL	*Dynamic Link Library*
DWT	*Discrete Wavelets Transform*
HTML	*Hyper Text Markup Language*
IoT	*Internet of Things*
JSON	*JavaScript Object Notation*
LOC	*Lines of Code*
MAC	*Media Access Control*
MD5	*Message-Digest Algorithm 5*
MSE	*Mean Squared Error*
PE	*Portable Executable*
RAM	*Random Access Memory*
SHA-1	*Secure Hashing Algorithm 1*
SHA-2	*Secure Hashing Algorithm 2*
SSIM	*Structural Similarity Index Matrices*
UPX	*Ultimate Packer for Executables*
VM	*Virtual Machine*
VT	*VirusTotal*

Capítulo 1

INTRODUÇÃO

No mundo moderno, o número de dispositivos conectados à Internet não para de aumentar, assim como a quantidade de serviços disponibilizados pela rede. Pesquisas indicam que o total de dispositivos inteligentes, pertencentes à IoT[1], chegará a 38,5 bilhões em 2020 [1]. Praticidade, agilidade e maior alcance nas relações de negócio são apenas algumas das vantagens encontradas por usuários de serviços e empresas ao utilizarem sistemas conectados à rede global. No entanto, tamanha difusão e quantidade de dispositivos conectados desperta também o interesse de criminosos que possuem o intuito de explorar as vulnerabilidades que acompanham esse crescimento. De acordo com um estudo da Nokia, houve um aumento de 400% no total de infecções a smartphones em 2016 [2].

A Segurança da Informação tem se tornado uma área cada vez mais crítica e investimentos bilionários são feitos em busca de proteção contra crackers[2]. É esperado que o investimento global supere os 93 bilhões de dólares nesta área, considerando tanto produtos de defesa como serviços de consultoria, para o ano de 2018 [3, 4] (Fig. 1.1).

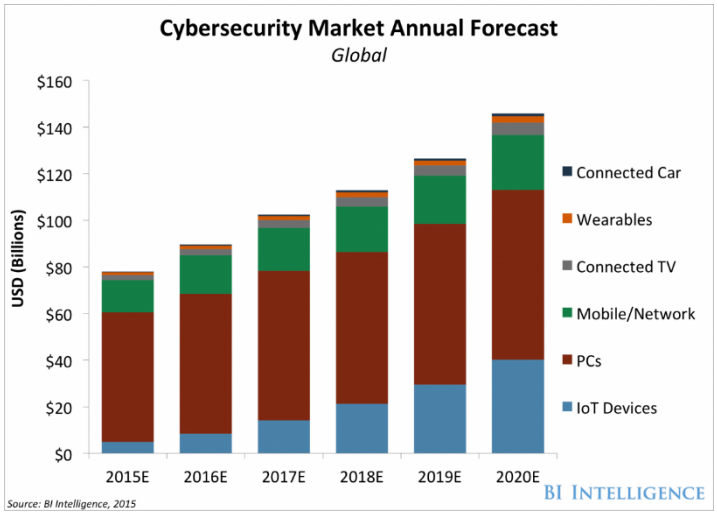

Figura 1.1: Investimentos globais anuais na área de Cyber Segurança [4].

[1]Termo designado para a conexão de itens do dia a dia e dispositivos inteligentes à Internet.
[2]Aquele que pratica a quebra, ou *cracking*, de um sistema de segurança.

Contudo, enquanto crescentes investimentos são realizados em busca de maior segurança da informação, novas e mais diversas técnicas criminosas também são desenvolvidas. A cada dia, surgem milhares de novos tipos de *malwares* e meios que objetivam dificultar a análise destes. Somente no período entre os anos de 2014 e 2018, o número total de *malwares* cresceu 136% [5], como apresentado na Fig. 1.2.

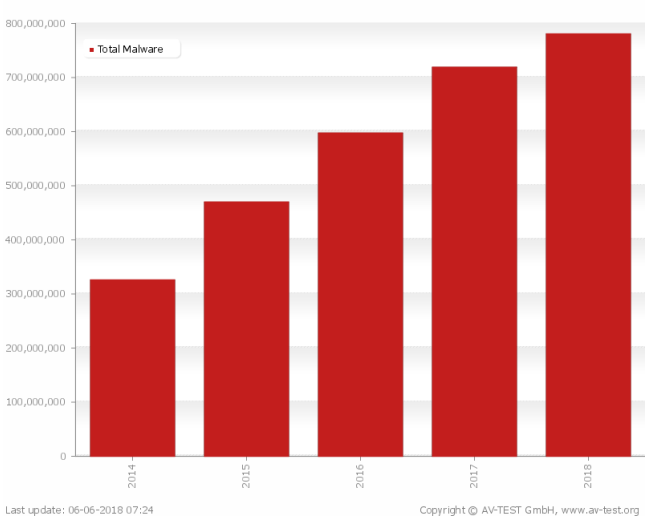

Figura 1.2: Quantidade total de *malwares* [5]

Além do volume, há também um aumento na robustez dos ataques, evidenciado em Fig. 1.3. Casos como o *ransomware WannaCry* e o *worm Stuxnet*, que visou obter o controle do sistema operacional de centrífugas de enriquecimento de urânio [6], servem de alerta para o assunto. E a construção de mecanismos de defesa requer previamente um adequado estudo e dissecamento dessas ameaças, a fim de entender seus modos de funcionamento e seus alvos. Basicamente, uma amostra de *malware* pode ser estudada aplicando-se técnicas de análise estática ou dinâmica, sejam estas básicas ou avançadas, de acordo com o objetivo e nível de ameaça [7].

Figura 1.3: Robustez crescente dos ataques maliciosos realizados no mundo.

Portanto, é necessário que investimentos na área de Segurança sejam feitos de forma progressiva, de modo que as ferramentas atuais continuem evoluindo e outras novas possam surgir. Diferentes técnicas de prevenção devem ser experimentadas para evitar prejuízos ou ao menos mitigar danos causados por ataques. Idealmente, são necessárias técnicas que identifiquem os *malwares* antes que eles causem qualquer dano na rede, podendo evitar até que cheguem aos dispositivos a ela conectados. Em 2017, o custo dos cibercrimes foi estimado em 600 bilhões de dólares, um número proveninente da realização de um estudo conduzido pela McAfee em 2018 [8]. E, apesar dos maiores esforços, o impacto econômico não está reduzindo. Isto se deve à diversos fatores, como por exemplo: novos usuários conectados à internet com práticas fracas de segurança; fator recompensa para o cibercrime em diversos países, seja por leis fracas ou facilidade de realizar ataques; complexidade de sistemas e equipamentos maiores, deixando margens para brechas a serem exploradas; anonimato permitido pelas criptomoedas para obter as rendas; dentre outros.

Region (World Bank)	Region GDP (USD, trillions)	Cybercrime Cost (USD, billions)	Cybercrime Loss (% GDP)
North America	20.2	140 to 175	0.69 to 0.87%
Europe and Central Asia	20.3	160 to 180	0.79 to 0.89%
East Asia & the Pacific	22.5	120 to 200	0.53 to 0.89%
South Asia	2.9	7 to 15	0.24 to 0.52%
Latin America and the Caribbean	5.3	15 to 30	0.28 to 0.57%
Sub-Saharan Africa	1.5	1 to 3	0.07 to 0.20%
MENA	3.1	2 to 5	0.06 to 0.16%
World	**$75.8**	**$445 to $608**	**0.59 to 0.80%**

Figura 1.4: Distribuição regional do cibercrime [8].

1.1 MOTIVAÇÃO

De acordo com os assuntos abordados na introdução deste trabalho, verifica-se que programas maliciosos se proliferam a taxas altíssimas, cerca de 300 mil novos diariamente [8], e se tornam cada vez mais complexos. Portanto, deve-se envidar o máximo de esforços em busca de novos meios de detecção, prevenção e estudo destes códigos. Será abordado no trabalho que mecanismos tradicionais e confiáveis de detecção podem ser enganados, falhando em detectar algumas amostras. Novas técnicas, como as aqui apresentadas, poderão auxiliar analistas de segurança na inspeção de *malwares* e possibilitar que mecanismos de defesa sejam mais rapidamente construídos.

Normalmente, o ser humano sente-se mais confortável para trabalhar com gráficos e imagens do que com binários e instruções *assembly*[3]. Por este motivo, é interessante utilizar táticas de análise estática que incluem representações visuais do código binário. Este tipo de técnica será discutida posteriormente neste trabalho.

1.2 OBJETIVOS

É objetivo deste trabalho realizar a comparação visual entre representações 2D do código binário de *malwares* reconhecidos, de arquivos duvidosos e de arquivos limpos, buscando identificar os arquivos maliciosos e detectar aqueles arquivos duvidosos que sejam *malwares*. É esperada a identificação de padrões que auxiliarão na detecção mais ágil dos códigos maliciosos e até mesmo detectá-los quando outros mecanismos não o fariam. Podem ser destacados os seguintes pontos:

- Reunir amostras de programas maliciosos e arquivos limpos;
- Representar de forma visual a assinatura do binário destes;
- Desenvolver algoritmos para a comparação entre as imagens obtidas;
- Criar base de dados com representações visuais classificadas por famílias e tipos *malwares*;
- Classificar amostras maliciosas;
- Detectar maliciosidade em amostras duvidosas e arquivos limpos;
- Verificar a eficácia do método proposto em diferenciar os diferentes tipos de *malwares* existentes;
- Verificar a eficácia na detecção de *malwares* não reconhecidos pelo antivírus escolhido.

1.3 METODOLOGIA

Primeiramente serão obtidas, por meio de *honeypots*[4] e repositórios próprios para estudo[5], as amostras de códigos maliciosos. Destas, serão eliminadas as que não forem detectadas por nenhum dos 60 antivírus da plataforma VirusTotal e as que não apresentarem relatório de escaneamento do antivírus escolhido como padrão, além dos arquivos de texto puro.

[3]Linguagem de programação de baixo nível
[4]São ferramentas que simulam sistemas com falhas conhecidas, visando atrair ataques e capturar *malwares*.
[5]https://github.com/ytisf/theZoo

Então será realizado um estudo acerca das possibilidades de visualização dos binários[6] de *malwares*, para em seguida buscar meios de compará-las. Desenvolvidos algoritmos que implementem essas formas de comparação, os mesmos serão testados sob diferentes configurações de modo a encontrar um ponto ótimo de operação. De forma resumida:

1. Obter amostras de *malwares*;
2. Eliminar os arquivos de texto puro, os que não foram submetidos ao escaneamento do antivírus escolhido e os que não forem detectados por nenhum AV da plataforma VirusTotal;
3. Classificar os *malwares* restantes de acordo com a rotulação do AV escolhido;
4. Gerar representações visuais dos binários destes arquivos;
5. Comparar estas imagens utilizando algoritmos de *hashing* perceptivo;
6. Comparar estas imagens utilizando algoritmos de comparação de imagens;
7. Repetir passos anteriores para duas resoluções adicionais;
8. Ajustar o algoritmo de modo a melhorar a taxa de detecção.
9. Avaliar e apresentar os resultados.

1.4 ESTRUTURA DO TRABALHO

No Capítulo 2, Fundamentação Teórica, será feita a abordagem de todo o pressuposto teórico necessário para a compreensão deste trabalho.

No Capítulo 3, Métodos Propostos, serão apresentadas as amostras *malware* e representações visuais de seus binários, ferramentas utilizadas, técnicas de interpretação e comparação das imagens obtidas, e algoritmos desenvolvidos.

No Capítulo 4, Resultado Experimentais, será feita a análise dos resultados e comparada a eficácia dos métodos utilizados na classificação das amostras.

Por último, o Capítulo 5, Conclusão, contém as principais conclusões obtidas com as simulações e os resultados analisados.

[6]Binários são arquivos executáveis ou não, que não sejam texto puro, estejam codificados e prontos para serem carregados ou interpretados por um programa ou *hardware*.

Capítulo 2

FUNDAMENTAÇÃO TEÓRICA

É realizada neste capítulo a apresentação dos conceitos e fundamentos teóricos aplicados no trabalho. Esse embasamento se mostra de vital importância, visto que é necessário para a completa ambientação com os termos e técnicas aqui utilizadas para o logro dos objetivos estipulados.

2.1 *MALWARES*

Software malicioso, ou *malware*, é todo o tipo de *software* que age com o objetivo de danificar, obter o controle ou vantagens indevidas de computadores, equipamentos ou de uma rede. Existem muitas variações com diferentes objetivos. Esses códigos maliciosos se valem de diferentes brechas e metodologias para atingirem seu objetivo de forma rápida e, na maioria das vezes, silenciosa. Podem ser citados entre os principais tipos de *malwares*: vírus, worm, *adware*, rootkit, bot e trojan (cavalo de Tróia).

Existem softwares que combatem esses códigos maliciosos, como os antivírus e *antispywares*. Porém, as atualizações de suas definições são mais lentas do que a criação de novas variantes de *malwares*; dentre as várias técnicas as quais os aplicativos antivírus utilizam, a mais comum é baseada na detecção da assinatura do arquivo suspeito ao comparar com um banco de dados previamente atualizado.

O Instituto SANS, uma companhia norte-americana de segurança, recomenda uma conduta de seis passos para responder a incidentes em computadores: preparação, identificação, contenção, erradicação, recuperação e lições aprendidas [9]. Consensos e cartilhas com ações como essas são criadas por empresas de segurança para que os incidentes sejam sanados no menor tempo e com a menor perda de dinheiro possível.

Malwares podem ser vistos como softwares extremamente eficientes em seu objetivo, e uma das métricas que permite afirmar isso é a quantidade de linhas de código que os definem. Larry Constantine observou que o *worm Stuxnet* tem cerca de 15 mil linhas de código - ou LOC, *lines of code* - para atacar os complexos sistemas operacionais SCADA da fabricante

Siemens em centrífugas nucleares iranianas [6]. Um arquivo de pouco mais do que meio *megabyte*. Esse número é muito maior que o que alguns autores estimam como valor médio de LOC para *malwares* típicos, de apenas 125 linhas como apresentado pelo *hacker* Peiter "Mudge" Zatko no evento *Black Hat* de 2011 [10].

Por outro lado, aplicações de defesa contra essas ameaças requerem um esforço, em termos de LOC, muitíssimo maior. Zatko estimou a quantia de 10 milhões de linhas de código para um serviço de defesa moderno. Como exemplo de aplicação legítima, o editor de texto *GNOME* para sistemas Linux engloba cerca de 70 mil linhas de código. Sistemas operacionais inteiros, como o já antigo e não mais suportado Windows XP, têm estimativas em 45 milhões de LOC [7]. Esses números mostram como as ameaças são eficientes e perigosas mesmo sendo tão "simples" em relação aos seus alvos.

2.1.1 Tipos de *malwares*

Malwares podem ser classificados de acordo com seu comportamento, como são instalados e como se propagam. Assim, são divididos em 9 tipos principais e mais comuns. Porém, muitos deles podem englobar em seu código diferentes comportamentos de variadas categorias de *malware*, seja para enganar programas encarregados da segurança ou para causar mais danos. Existem milhares de variantes dentre cada classificação, gerando diversas famílias com semelhanças estruturais. Os programas maliciosos também podem ser divididos através da dependência ou não do hospedeiro. Vírus, *backdoors e trojans*, por exemplo, necessitam de um código hospedeiro, diferentemente dos *worms*. Outra forma de classificá-los é quanto à capacidade de replicação. *Worms* e vírus são capazes de se replicarem, ao contrário dos outros tipos de *malwares*. Porém, é cada vez mais fácil encontrar malwares com características distintas, possuindo capacidades híbridas. Isso ocorre para que os códigos infecciosos tenham maiores capacidades de ataque e devastação. As classificações a seguir buscam abordar de forma geral o comportamento dos *malwares* existentes.

2.1.1.1 Vírus

É um dos tipos mais antigos de código malicioso. Está presente na internet desde os seus primórdios. Basicamente, é um arquivo infectado com um código que, ao entrar em operação, se replica e camufla entre os arquivos do computador. Realiza exatamente o que foi programado em seu código. Normalmente, precisa da interação humana para entrar em ação através da sua execução. São de difícil detecção sem a ajuda de programas antivírus instalados no dispositivo. Podem se esconder estando anexados a qualquer tipo de arquivo que possa ser executado por um usuário, como arquivos .exe, .bat e até protetores de tela .scr. Além da possibilidade de roubarem informações, outro importante dano é a utilização de recursos computacionais do dispositivo. O meio mais comum para a propagação destes é através de mídias removíveis, os *flashdrives* [7, 11, 12].

2.1.1.2 *Worms*

Os *worms* são softwares contidos em seus próprios containers, isto é, podem não depender de nenhum outro arquivo para se propagarem. Multiplicam-se automaticamente, explorando brechas na rede e falhas de seguranças nos equipamentos. Acessam e destroem arquivos do disco rígido. Continuam em ação até corromperem o máximo de dados possível. Existem versões que não danificam arquivos, apenas multiplicando-se indefinidamente. Isto não é um risco menor, já que o aumento exacerbado do tráfego de rede pode ser extremamente prejudicial para as mais diversas operações de um mundo cada vez mais conectado [7, 11, 12].

2.1.1.3 *Adware*

Nome dado a programas designados para exibir anúncios no computador ou dispositivo, coletar informações de marketing do usuário e redirecionar endereços digitados para páginas da web pré-definidas. São softwares que atuam com o objetivo de gerar o máximo de renda possível através de anúncios. São considerados maliciosos quando coletam dados de forma oculta ao usuário, apresentando anúncios extremamente direcionados e convenientes [7, 11, 12].

2.1.1.4 *Spyware*

O *spyware* é um código malicioso que visa obter informações sobre o funcionamento do dispositivo e atividade do usuário. Pode ser utilizado tanto de forma legítima como maliciosa. Usuários podem instalar *spywares* de forma legítima a fim de monitorar computadores próprios e evitar que terceiros façam uso não autorizado ou abusivo. Dados como sítios visitados, teclas pressionadas, e-mails enviados ou recebidos, programas executados, dentre outros, são enviados para o atacante. De forma maliciosa, é utilizado para espionar o usuário em questão, daí a origem do seu nome. Uma variação de *spyware* é o *keylogger, malware* especializado em roubar, especificamente, o conteúdo digitado [7, 11, 12].

2.1.1.5 *Trojan Horse* (cavalo de tróia)

O cavalo de Tróia, como o próprio nome sugere, são códigos que se apresentam para o usuário como úteis, escondendo as reais intenções até que sejam devidamente instalados. Aparentemente, realizam as funções para as quais foram programados, realizando também uma série de outras atividades sem o conhecimento do usuário. É possível encontrar até softwares antivírus falsos que na verdade são cavalos de Tróia.

Como a lenda na qual é inspirado, uma vez transferido para "dentro" do computador e executado, é extremamente perigoso. Algumas de suas variantes buscam encontrar *backdo-*

ors para entregar o acesso e controle do dispositivo a terceiros, deixando os dados do usuário expostos. Outras, por sua vez, buscam destruir e apagar dados do disco rígido [7, 11, 12].

2.1.1.6 *Rootkit*

Conjunto de ferramentas que permitem esconder a presença de um invasor ou outros códigos maliciosos no equipamento comprometido, dificultando bastante a detecção das ameaças. São instalados e não se replicam automaticamente. Uma vez instalados, operam nos mais baixos níveis do sistema, tornando sua remoção uma tarefa complicada. É comum ser necessário apagar discos rígidos completamente para eliminar uma ameaça *rootkit* [7, 11, 2].

2.1.1.7 *Backdoor*

Este tipo de *malware* é especializado em encontrar falhas de segurança em *firewalls*, sistemas e programas, se instalando e permitindo a terceiros o acesso ao dispositivo através destas falhas. Após instalado, ele garante a possibilidade de o invasor retornar ao equipamento invadido sem a necessidade de um novo ataque. É muito perigoso, pois pode criar serviços que gerem brechas no sistema hospedeiro com a finalidade de ser o ponto de entrada de diversos *malwares*. Usualmente, este código malicioso é adicionado a serviços de conexão remota que substituem os originais. Também pode se referir ao ato de códigos maliciosos incluídos em determinados *softwares* por fabricantes sob o pretexto de necessidades administrativas, expondo a privacidade de usuários a riscos [7, 11, 12].

2.1.1.8 *Ransomware*

O *ransomware* é um tipo de malware que infecta o computador e restringe o acesso do usuário aos seus próprios dados. O disco rígido (ou seções deste) é criptografado e a chave é liberada apenas com o pagamento de um resgate. Alguns especialistas em segurança consideram este tipo de *malware* como o mais perigoso. Ainda que a extorsão digital esteja presente como forma de ataque desde a década de 80, com o advento e explosão das moedas digitais por volta de 2013 o *ransomware* ganhou impulsão total. Os criminosos agora podem coletar suas recompensas de forma mais rápida e anônima com as criptomoedas. É a categoria de *malware* que cresce mais rapidamente, pois se tornou altamente lucrativa. Milhões de usuários domésticos e empresas infectadas representam uma renda potencialmente bilionária, enquanto os riscos são baixos. Um estudo do FBI apontou que foram pagos 209 milhões de dólares em resgates no primeiro quadrimestre de 2016; comparando com 2015, haviam sido pagos 24 milhões de dólares em resgate no ano inteiro [7, 11, 12, 13].

2.1.1.9 *Bots*

Malwares com funcionamento similar aos *worms*, porém com mecanismos adicionais para a comunicação com o invasor. Isto permite que os equipamentos infectados sejam controlados à distância, criando assim as *botnets*. *Botnets* representam um grande problema na atualidade, dado que cada vez mais equipamentos são conectados à internet e muitos desses não receberam as devidas precauções de segurança, como alterar usuário e senha padrões em um roteador. Com isso, diversos equipamentos passam a compor uma *botnet* sem que seus respectivos usuários tenham a ciência do fato. Isto representa um enorme campo explorado por *crackers*, que criam e controlam redes zumbis essencialmente para DDoS cada vez mais robustos [7, 11, 12, 14].

2.1.1.10 *Downloaders*

Esta categoria de *malware* é especializada em buscar e manter brechas de segurança em um dispositivo unicamente para efetuar o *download* de diversos outros códigos maliciosos. O *downloader* é instalado pelo *cracker* assim que o acesso ao sistema é estabelecido [7, 11, 12].

2.2 ANÁLISE DE *MALWARE*

O processo de identificação de *malwares*, de acordo com [7, 15], consiste nos seguintes estágios: preservação forense e análise dos dados voláteis; análise da memória do sistema infectado; exame das mídias de armazenamento permanentes; análise das características do arquivo suspeito; e por fim, análise estática e dinâmica deste.

Um bom modo de prevenir e reduzir danos causados por *malwares* é entendê-los em suas diferentes formas e seus objetivos. Para compreendê-los, entender os mecanismos de funcionamento, sinais e rastros deixados, faz-se necessária a sua análise. De posse do arquivo suspeito e buscando identificá-lo e classificá-lo, é possível utilizar diversas ferramentas e artifícios para descobrir como o código malicioso opera.

Visualizadores do binário, *disassemblers* e Engenharia Reversa são algumas das técnicas possíveis de serem aplicadas para a obtenção de novos detalhes e informações. Fundamentalmente, existem duas abordagens para a análise de malwares: a estática e a dinâmica. Na primeira, o código é analisado sem que seja executado. Na segunda, o código malicioso é executado em um ambiente controlado, e então, é possível chegar a importantes conclusões sobre seu funcionamento e efeitos.

2.2.1 Análise estática

A análise estática consiste em examinar o arquivo sem, efetivamente, executar as suas instruções. Pode ser utilizada para gerar rápidas conclusões, se é um arquivo infectado ou se está ofuscado, por exemplo. Padrões podem ser identificados através de assinaturas geradas. Diversas técnicas para a análise estática são discorridas em [7]. Tratar-se-á de algumas delas nesta seção.

Dentre as ferramentas que auxiliam nessa tarefa, estão os *disassemblers*. Estes softwares analisam códigos executáveis e apresentam o código em *assembly* utilizado para realizar aquela compilação. Assim, é possível analisar o código à procura de *opcodes*[1] ou elementos suspeitos, isto é, se o executável em questão não estiver empacotado. IDA Pro[2], OllyDbg[3] são exemplos deste tipo de software. Em [16] os autores utilizaram o código malicioso disassembled para identificar *opcodes* e gerar assinaturas específicas para cada comportamento. Porém, esta técnica de análise estática é considerada avançada e requer um amplo conhecimento sobre a programação em *assembly* e como os códigos interagem com o sistema operacional.

Uma técnica comum para identificação das amostras é o *hashing*, que consiste em obter a *hash*, um valor único, gerada por um programa. Funciona como uma impressão digital, em que cada *malware* e software possuem assinaturas únicas. Isto será melhor definido em seções posteriores deste trabalho. De posse destes valores de chave únicos, pode ser feita a busca em sites que agregam informações relacionadas aos antivírus e detecções de *malwares*, como por exemplo o site VirusTotal[4], para descobrir se o arquivo é um código malicioso conhecido. Uma limitação deste método é que essas assinaturas são modificadas a cada alteração realizada no código. Diante disto, *crackers* estão sempre criando novas variantes mesmo de códigos maliciosos conhecidos. Pequenas alterações no código, encapsulamento, criptografia ou ofuscação são técnicas utilizadas para dificultar o trabalho de identificação destes códigos maliciosos. Isto gera um atraso até que essas novas assinaturas sejam identificadas como maliciosas e acrescentadas aos bancos de dados dos *softwares* antivírus [7].

É possível ainda utilizar aplicativos que obtenham informações acerca das *strings*, DLLs e estrutura do arquivo executável, como no formato executável PE. A estrutura PE será brevemente discutida ao longo deste trabalho. Estes aplicativos são conhecidos por fornecerem um *Initial Malware Assessment* — traduzindo, Assessoria Inicial de Códigos Maliciosos. *Strings* são sequências de caracteres de um programa e DLLs são bibliotecas utilizadas por executáveis no sistema Windows. Algumas importações de bibliotecas logo entregam que a finalidade do arquivo em questão é suspeita, assim como a quantidade de DLLs utilizadas; muito poucas podem indicar um empacotamento, enquanto muitas podem indicar que o código possui finalidades ocultas. Podem ser citados como exemplos de *softwares* úteis para

[1] Instruções em *assembly* que contêm o código de operação para que o processador funcione.
[2] https://www.hex-rays.com/products/ida/
[3] http://www.ollydbg.de
[4] https://www.virustotal.com/pt/

uma análise inicial do malware o Pestudio[5] (Fig. 2.1) e o Portex Analyzer[6].

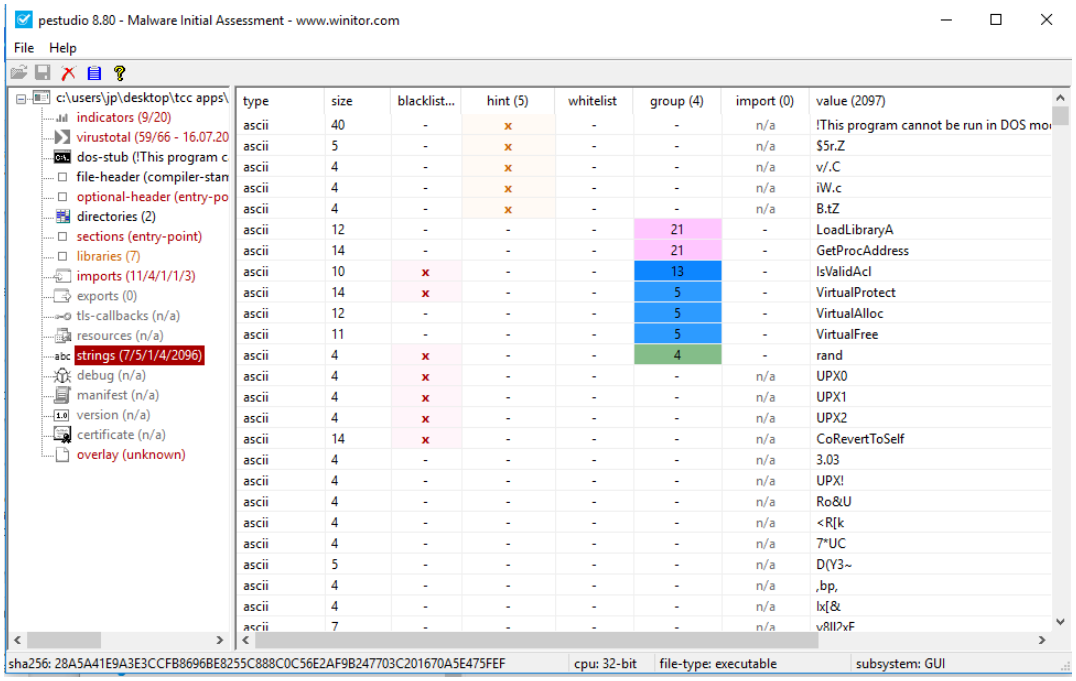

Figura 2.1: Captura de tela do PEStudio mostrando as *Strings* encontradas dentro do arquivo.

2.2.2 Análise dinâmica

Este tipo de análise envolve a execução das amostras em um ambiente adequado. Com isso, é possível obter informações não disponíveis com a análise estática. O funcionamento do *malware* é analisado através de dumps da memória volátil, arquivos do sistema modificados, sniffing da rede para detectar informações transmitidas e recebidas. Assim, é possível observar o real funcionamento do *malware*, que pode ter sido codificado com strings e métodos extras no binário para ludibriar analistas. É comum que a amostra seja executada em uma *Sandbox*, isto é, um ambiente virtualizado adequado e facilmente reversível ao estado original. Existem *sandboxes* onlines e *standalones*. Da primeira categoria, podem ser citados Malwr[7] e *ThreatTrack ThreatAnalyser*[8]. As *sandboxes standalones* são executadas no equipamento físico do usuário, como a Cuckoo Sandbox[9], o VirtualBox[10] e VMware[11]. Como observado no trabalho de Leite [17], diversas análises aplicadas conjuntamente com *sandboxes* permitem a obtenção de características da execução do *malware*.

Uma das vantagens deste método é a possibilidade de analisar *dumps* da memória volátil do dispositivo e alterações feitas no registro do Windows, conforme visto em Fig. 2.2.

[5]https://www.winitor.com
[6]https://github.com/katjahahn/PortEx
[7]https://malwr.com
[8]https://www.threattrack.com/malware-analysis.aspx
[9]https://cuckoosandbox.org
[10]https://www.virtualbox.org
[11]https://www.vmware.com/br.html

Malwares podem armazenar importantes dados e deixar traços de seu funcionamento na memória RAM. Um risco deste tipo de análise é a evasão de amostras maliciosas sofisticadas, podendo infectar outros dispositivos que estejam conectados à mesma rede. Outra desvantagem é a necessidade de avaliação por parte de analistas das evidências coletadas no ambiente virtual, assim como dos relatórios gerados no caso de *softwares* como o Cuckoo.

```
user@ubuntu:~$ vol -f memory_images/example.vmem --profile=WinXPSP2x86 pslist
Volatile Systems Volatility Framework 2.3_alpha
Offset(V)  Name                    PID   PPID   Thds     Hnds   Sess  Wow64 Start                Exit
---------- -------------------- ------ ------ ------ -------- ------ ------ -------------------- ----------
0x819cc830 System                    4      0     60      209 ------      0
0x818efda0 smss.exe                384      4      3       19 ------      0 2011-09-26 01:33:32
0x81616ab8 csrss.exe               612    384     12      473      0      0 2011-09-26 01:33:35
0x814c9b40 winlogon.exe            636    384     16      498      0      0 2011-09-26 01:33:35
0x81794d08 services.exe            680    636     15      271      0      0 2011-09-26 01:33:35
0x814a2cd0 lsass.exe               692    636     24      356      0      0 2011-09-26 01:33:35
0x815c2630 vmacthlp.exe            852    680      1       25      0      0 2011-09-26 01:33:35
0x81470020 svchost.exe             868    680     17      199      0      0 2011-09-26 01:33:35
0x818b5248 svchost.exe             944    680     11      274      0      0 2011-09-26 01:33:36
0x813a0458 MsMpEng.exe            1040    680     16      322      0      0 2011-09-26 01:33:36
0x816b7020 svchost.exe            1076    680     87     1477      0      0 2011-09-26 01:33:36
0x817f7548 svchost.exe            1200    680      6       81      0      0 2011-09-26 01:33:37
0x8169a1d0 svchost.exe            1336    680     14      172      0      0 2011-09-26 01:33:37
0x813685e0 spoolsv.exe            1516    680     14      159      0      0 2011-09-26 01:33:39
0x818f5cd0 explorer.exe           1752   1696     32      680      0      0 2011-09-26 01:33:45
0x815c9638 svchost.exe            1812    680      4      102      0      0 2011-09-26 01:33:46
0x8192d7f0 VMwareTray.exe         1876   1752      3       84      0      0 2011-09-26 01:33:46
0x818f6458 VMwareUser.exe         1888   1752      9      245      0      0 2011-09-26 01:33:47
0x8164a020 msseces.exe            1900   1752     11      205      0      0 2011-09-26 01:33:47
0x81717370 ctfmon.exe             1912   1752      3       93      0      0 2011-09-26 01:33:47
0x813a5b28 svchost.exe            2000    680      6      119      0      0 2011-09-26 01:33:47
0x81336638 vmtoolsd.exe            200    680      5      234      0      0 2011-09-26 01:33:47
0x81329b28 VMUpgradeHelper         424    680      5      100      0      0 2011-09-26 01:33:48
0x812d6020 wscntfy.exe            2028   1076      3       63      0      0 2011-09-26 01:33:55
0x812c1718 TPAutoConnSvc.e        2068    680      5       99      0      0 2011-09-26 01:33:55
0x812b03e0 alg.exe                2272    680      7      112      0      0 2011-09-26 01:33:55
0x81324020 TPAutoConnect.e        3372   2068      3       90      0      0 2011-09-26 01:33:59
0x814e7b38 msiexec.exe            2396    680      5      127      0      0 2011-09-26 01:34:45
0x814db608 cmd.exe                3756   1752      3       56      0      0 2011-09-30 00:20:44
0x812f59a8 cmd.exe                3128    200      0 --------      0      0 2011-09-30 00:26:30  2011-09-30 00:26:30
```

Figura 2.2: Análise do *dump* de memória com o framework Volatility. Processo *explorer.exe* suspeito [18].

2.2.3 Técnicas anti-análise e anti-detecção

Frequentemente, os criadores de *malwares* buscam dificultar a tarefa de detecção e análise de seus códigos. É objetivo destes indivíduos que não seja possível ou que não seja fácil realizar a engenharia reversa de seus códigos. Empacotamento ou *packing*, ofuscação e encriptação são alguns dos métodos utilizados para tal finalidade [7]. Existem ainda técnicas inteligentes que impedem a análise do *malware* mesmo em ambientes dinâmicos. Pode-se citar nesta categoria técnicas Anti-máquinas virtuais, anti-emulação, *anti-debugging* e *anti-disassembly*. O conjunto destas técnicas dificulta o trabalho de analistas de segurança, elevando os níveis de conhecimento e tempo necessários para o estudo dos arquivos infecciosos.

2.2.3.1 Ofuscação e empacotamento de *malware*

É o ato de modificar o código do exemplar para dificultar sua análise e detecção. Para isto, são utilizados os *packers*, softwares que comprimem, transformam ou criptografam o

código do executável desejado. Desta forma, é dificultada tanto a identificação dos maliciosos pelos antivírus, já que estes também se baseiam nas assinaturas contidas em uma base de dados, que variam de acordo com o código, como a análise por analistas de segurança. O funcionamento do *packer* consiste em modificar o arquivo e colocá-lo em um *software* envelope[19]. Existem quatro tipos mais comuns de *packers* (Fig. 2.3):

- Compressores: aplicam compressão no código desejado;
- *Crypters*: aplicam criptografia no alvo;
- Protetores: aplicam criptografia e compressão;
- *Bundlers*: colocam vários códigos juntos dentro de um único envelope de software.

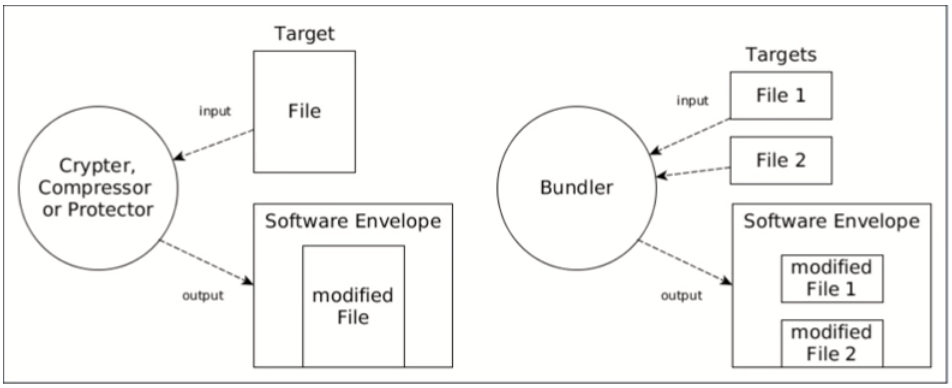

Figura 2.3: Categorias de *packers* [20].

Alguns indicativos que podem ser observados através da análise estática e permitem a conclusão de que se trata de um arquivo ofuscado: pouquíssimos *imports* realizados pelo aplicativo, funções *LoadLibraryA* e *GetProcAddress* importadas, poucas *strings* úteis visualizáveis no binário com ferramentas de análise de PEs, alta entropia na representação visual do binário, dentre outras [20].

Estas técnicas dificultam a análise estática dos maliciosos, sendo necessário efetuar o processo reverso ou uma análise dinâmica para a obtenção de mais informações relevantes, o que nem sempre é possível por ser necessário preparar um ambiente adequado. Por vezes, são utilizados *packers* comuns, como o UPX[12]. Nestes casos, existem *unpackers* que trazem os códigos novamente ao estado inicial. Quando o *cracker* deseja tornar esse processo de reversão ainda mais trabalhoso, algo que acontece na maioria das vezes, pode-se aplicar técnicas customizadas para realizar a compressão ou criptografia de seu código.

2.2.3.2 Anti-máquina virtual

Existem amostras que utilizam avançadas técnicas que impedem sua análise em ambiente virtual, um comum procedimento durante uma análise dinâmica. Para isso, é realizada a

[12]https://upx.github.io

detecção de traços, artefatos que são deixados por VMs quando executadas. Estes traços podem incluir processos típicos sendo executados no sistema, chaves no registro ou pastas no disco rígido. A criação de placas de rede virtualizadas também pode contribuir para que um *malware* detecte a sua execução em ambiente virtualizado. Os 3 primeiros bytes de endereços MAC estão tipicamente relacionados ao fabricante do dispositivo de rede. Por exemplo, 00:0C:29 está relacionado à VMware. Quando o *malware* constata que está sendo executado em um ambiente controlado, como uma máquina virtual, pode interromper sua execução, não apresentar funcionamento malicioso, ou ainda tentar escapar do ambiente virtual e infectar o sistema hospedeiro [7, p. 369-380]. As imagens 2.4 e 2.5 exemplificam alguns dos itens que podem ser detectados por estes códigos maliciosos especializados. Códigos maliciosos que buscarem por strings contendo "Vmware"nos processos em execução ou no registro do sistema podem encontrar resultados como os das imagens 2.4 e 2.5.

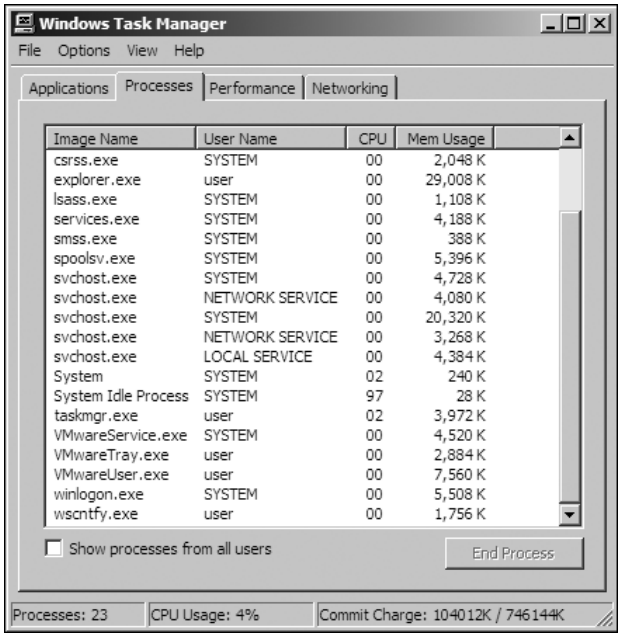

Figura 2.4: Processos listados em uma VM VMware [7].

```
[HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\Scsi\Scsi Port 0\Scsi Bus 0\Target Id 0\Logical Unit Id 0]
"Identifier"="VMware Virtual IDE Hard Drive"
"Type"="DiskPeripheral"

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Reinstall\0000]
"DeviceDesc"="VMware Accelerated AMD PCNet Adapter"
"DisplayName"="VMware Accelerated AMD PCNet Adapter"
"Mfg"="VMware, Inc."
"ProviderName"="VMware, Inc."

[HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Control\Class\{4D36E96F-E325-11CE-BFC1-08002BE10318}\0000]
"LocationInformationOverride"="plugged into PS/2 mouse port"
"InfPath"="oem13.inf"
"InfSection"="VMMouse"
"ProviderName"="VMware, Inc."
```

Figura 2.5: *Strings* relacionadas ao VMware encontradas no registro de uma VM [7].

2.2.3.3 Anti-*Disassembly*

O objetivo desta técnica é impedir que seja feito o *disassembly* automático por softwares utilizados por analistas de segurança, impedindo que qualquer código esteja disponível antes que o programa seja executado [21]. Com isso, análises estáticas são gravemente prejudicadas. A decompilação falha por consequente, assim como o processo de extração das *strings* não retorna muitos valores legíveis. Exemplos de métodos utilizados que compõem esta categoria:

- Maliciosos criptografados e polimórficos[13];
- Arquivos empacotados;
- Código gerado dinamicamente;
- Criptografia com diferentes chaves ou com diferentes algoritmos por diversas vezes.

2.3 TÉCNICAS DE VISUALIZAÇÃO DE DADOS APLICADAS À SEGURANÇA DA INFORMAÇÃO

A visualização de dados consiste em apresentar de forma gráfica um determinado conteúdo ou informação. Esta forma de representação traz novas possibilidades aos tomadores de decisão, que podem entender conceitos complicados mais facilmente ou identificar novos padrões, tarefas que seriam mais complicadas com os dados apresentados apenas como forma textual. Esse cenário pode ser aplicado aos analistas de segurança no campo da análise de malwares. Apenas realizar a leitura de binários e códigos *assembly* torna a análise estática uma atividade maçante e por vezes lenta. Por isso, é vital que novas técnicas sejam desenvolvidas neste sentido, de facilitar e agilizar a identificação de padrões e características de *malwares*.

Como observado nos trabalhos de [22, 23, 24], esta análise pode ser feita de variadas formas utilizando a geração de imagens a partir do código binário de exemplares maliciosos. Essas imagens geradas, representações visuais das assinaturas, podem variar quanto à forma como a informação e os dados são organizados graficamente. Podem ser citadas algumas técnicas: baseadas em pixel, geométricas, 3d, por ícones, *treemap* e dendrograma. Serão brevemente descritas a seguir as técnicas que mais foram aplicadas no trabalho. O livro escrito por Ma, Goodall e Conti [25] reúne diversas formas de visualização de dados para a área de Segurança da Informação, aprofundando este conteúdo com 18 textos oriundos de estudiosos da área.

[13] *Malwares* que estão constantemente modificando características com o intuito de dificultar a detecção. Nomes de variáveis, chaves de criptografia, rotinas de descriptografia são exemplos de itens possíveis de serem modificados, resultando numa assinatura variável.

2.3.1 Técnica baseada em pixel

Neste conjunto de técnicas, os atributos analisados são organizados em pequenos conjuntos, que por sua vez são organizados em arranjos. No caso estudado, os atributos são os bytes que compõem o código de um arquivo. Estes são organizados em pixels. São representados através dos pixels os valores dos atributos, sendo geradas diferentes cores para cada tipo de valor. Com o auxílio de um mapa de cores, é possível identificar padrões visuais na forma que os pixels são arranjados e coloridos. Diferentes ferramentas possuem variadas técnicas para alocação e coloração dos pixels, trazendo novas informações e perspectivas. Daniel Keim[26] apresentou diferentes curvas para o preenchimento do espaço bidimensional. Na Figura 2.6, um exemplo da representação pixel construída de forma sequencial, favorável para percepção das diferentes seções de um código de estrutura *Portable Executable*.

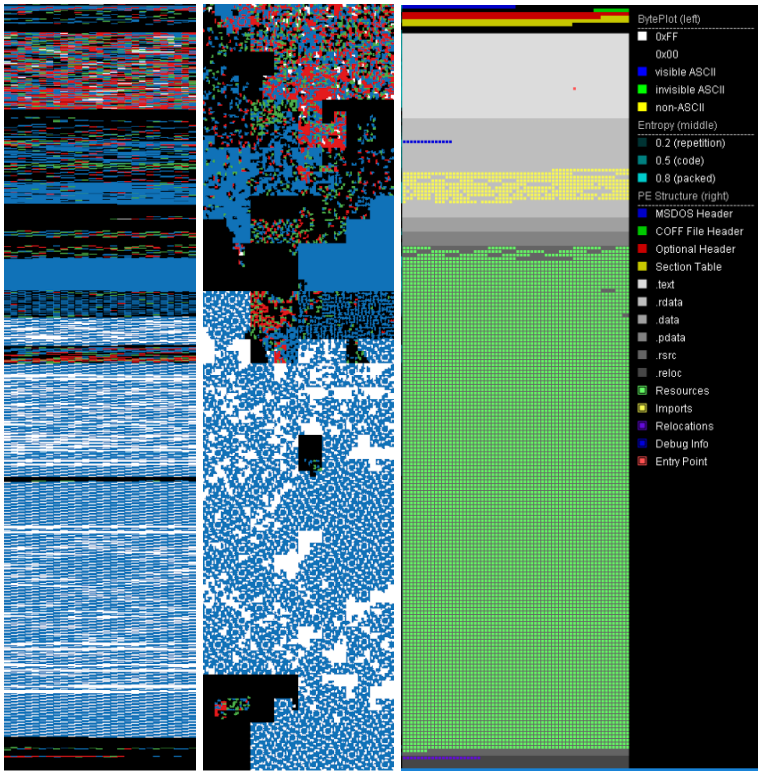

Figura 2.6: Análise Sequencial, *Cluster* e da Estrutura do executável Calc.exe.

2.3.2 Técnicas geométricas

Técnicas Geométricas de visualização de dados permitem rapidamente identificar padrões nos códigos analisados. São resultado de transformações geométricas dos valores dos seus atributos, como reorganizações e projeções [27]. Sequências de bytes comuns tendem a gerar marcas ou padrões visuais parecidos. Um método bastante utilizado aqui é através da matriz de espalhamento (*scatterplot matrix*, Fig. 2.7), resultando em uma matriz bidimensional na qual cada plotagem provém da combinação das demais dimensões. Desta forma,

correlações entre atributos sobressaltam visualmente. Outra possibilidade é o Mapa Auto-Organizável de Kohonen, como visto em [28].

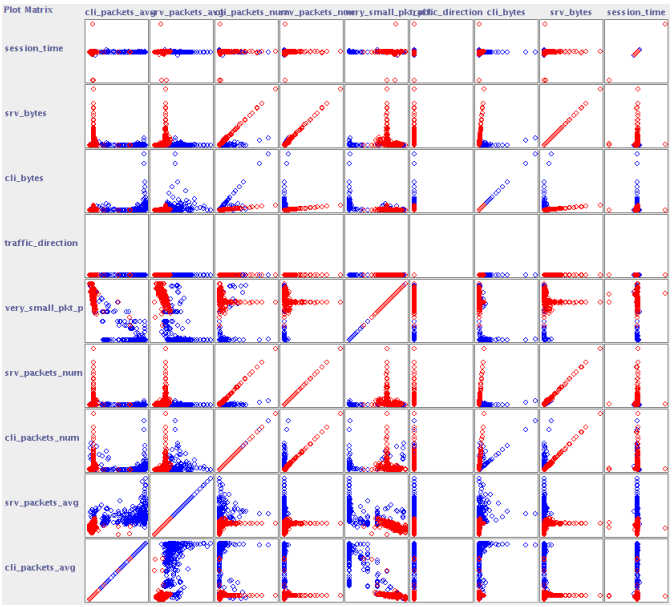

Figura 2.7: Matriz de espalhamento para visualização e comparação de tráfegos maliciosos e inofensivos.

2.3.3 Técnica tridimensional

Nesta categoria, são utilizadas ferramentas que geram imagens de forma semelhante à técnica pixel. A diferença está no arranjo da informação, realizado em três dimensões, como se pode notar em Fig. 2.8. Costumam ser formas interativas para a visualização dos dados, já que são gráficos relativamente complexos de serem analisados sob apenas uma perspectiva. Técnicas deste tipo podem apresentar-se extremamente poderosas e detalhadas, mas requerem uma longa ambientação do analista com os seus padrões e formas.

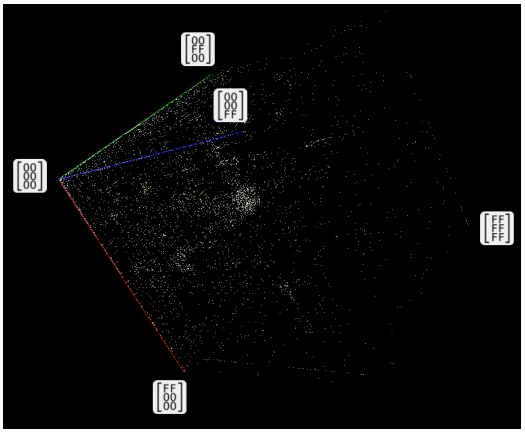

Figura 2.8: Cubo tridimensional para o aplicativo Calc.exe.

2.3.4 Técnicas baseadas em grafos

São utilizadas, principalmente, para demonstrar relações entre os objetos, representados por vértices. Diferentes técnicas nesta área modificam o posicionamento de vértices e arestas, de forma a evidenciar características específicas, como padrões e valores associados às relações entre os objetos. Na Figura 2.9, estão em evidência pontos coloridos com a cor vermelha, *malwares* da família Tsunami, os quais estão, quase que em sua totalidade, agrupados no mesmo local. Representações deste tipo são criadas com bastante frequência através da utilização da API JUNG[14].

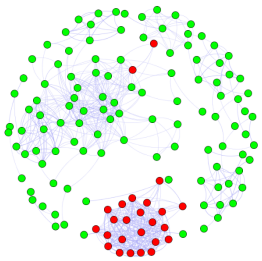

Figura 2.9: Nível de similaridade entre diferentes famílias *malwares* utilizando a técnica de Grafos.

2.3.5 Técnica *dot plot*

É uma técnica popular na bioinformática para a análise visual de genomas. Pode ser utilizada para o cálculo de auto similaridade em um arquivo ou para a medição de similaridade entre duas fontes diferentes. A representação é criada a partir de uma matriz da sequência de bytes do arquivo. *Strings* repetidas são destacadas por blocos maciços de pixels, enquanto seções com alta entropia são representadas por pontos escassos. A técnica é discutida em detalhes em [29] e utilizada na engenharia reversa de binários em [30].

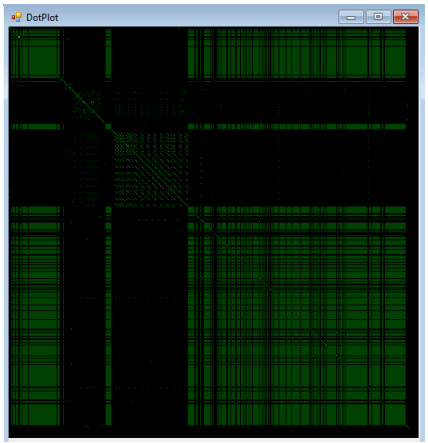

Figura 2.10: Representação dotplot para o Calc.exe

[14]http://jung.sourceforge.net

2.4 ENTROPIA DA INFORMAÇÃO

Claude Shannon é considerado o pai da Teoria da Informação, área desenvolvida inicialmente para o estudo da transmissão, compressão e armazenagem de dados. Um dos muitos pilares estabelecidos por Shannon neste campo foi a Entropia da Informação. Esta é a medida da falta de detalhamento da informação, como definido por Billouin em [31]. Shannon define a entropia H em [32] como:

$$H = -\sum_{i=1}^{n} \rho \cdot log \cdot \rho \tag{2.1}$$

A base do logaritmo é variável. É possível utilizá-la como base 256, representando o número de valores que um byte pode possuir. Isto gera valores de entropia no intervalo de [0,1]. Algumas outras fontes, como o site VirusTotal, utilizam a base 2, tomando como base o número de valores os quais um único bit pode tomar. Isto resulta em valores entre [0,8]. Quanto maior a informação, menor será a entropia. O que significa que no campo da análise de *malwares*, por exemplo, amostras empacotadas e criptografadas vão apresentar uma alta entropia. Este pode ser um importante indicador visual, gerando a possibilidade de rapidamente dispensar uma maior atenção a uma determinada seção durante a análise de um código que esteja sob suspeita de infecção.

2.5 ALGORITMOS DE DISPERSÃO UNIDIRECIONAL: FUNÇÕES DE *HASHING*

Funções *hash* são algoritmos matemáticos determinísticos que mapeiam dados de comprimento aleatório em saída de tamanho fixo em base hexadecimal, dispersando os bits de entrada de forma não correlacionada às mudanças. Ou seja, uma pequena mudança na entrada, seja um simples caractere em uma frase inteira, ou um pixel em uma foto, acarreta uma saída completamente diferente, sendo essa característica conhecida como Efeito Avalanche e observada na Fig. 2.11.

```
In [25]: hashlib.md5("teste").hexdigest()
Out[25]: '698dc19d489c4e4db73e28a713eab07b'

In [26]: hashlib.md5("testr").hexdigest()
Out[26]: 'efe8de352cd483af54ec60595508e54e'

In [27]: hashlib.md5("Teste").hexdigest()
Out[27]: '8e6f6f815b50f474cf0dc22d4f400725'
```

Figura 2.11: Demonstração do Efeito Avalanche utilizando função de *hash* criptográfica MD5 com linguagem Python

Sendo determinísticas, ocorrerá sempre a mesma saída caso não haja alteração da infor-

mação que alimente a função. Por ser um algoritmo unidirecional, não é possível obter a chave de entrada de posse apenas da *hash* calculada. Portanto, esses algoritmos podem ser utilizados na verificação de integridade de arquivos e sua identidade.

As aplicações para essas funções são diversas e abrangem desde a área de tecnologia da informação, à aplicações em correios eletrônicos, proteção de senhas, transferência de arquivos, criptografia da informação, assinatura de documentos e inclusive na proteção de propriedade intelectual e combate à pirataria.

Em criptografia de senhas, por exemplo, bons servidores não armazenam as senhas dos usuários em texto puro. São guardadas somente *hashes* geradas no momento de cadastramento do usuário, e em eventos de autenticação são comparadas estas guardadas com as calculadas de acordo com a entrada do cliente. No entanto, caso seja obtida uma base com sua lista de *hashes* e conhecido o algoritmo aplicado, um elemento mal intencionado pode comparar as entradas desta base com valores de *hashes* para palavras comuns — teste, 123, senha, etc. — em tabelas facilmente encontradas. Essas tabelas, chamadas de *rainbow-tables*, geralmente são obtidas por força bruta. Nota-se a necessidade de utilizar cifras e palavras-chave mais fortes para a autenticação [33].

Felizmente, existem mecanismos para se defender de ataques baseados em *rainbowtables*. Um deles é a Salgagem, ou *Salting*: adiciona-se uma *string* aleatória à entrada do usuário e então calcula-se o *hash* dessa nova palavra. Assim, armazena-se o valor de saída e essa *string* aleatória para cada entrada, impossibilitando que seja utilizada uma tabela pronta e forçando o atacante a recalcular cada palavra possível e comparar com as *hashes* do banco de dados por força bruta [33].

Em se tratando de integridade da informação, uma simples comparação entre valores *hash* de arquivo original e recebido permite confirmar ou não se transmissão foi confiável e segura, tendo sido criada uma cópia exata do original. É criada uma identificação única para aquela versão de arquivo, como uma digital (ou *fingerprinting*, em inglês), sendo bastante aplicada na transmissão de arquivos P2P (*peer to peer*) [34].

Entre os tipos de algoritmos de *hashing* mais conhecidos, os mais comuns são MD5 [35], com 128 bits e extensamente utilizado na verificação de integridade de arquivos, e o SHA-1 [36], com 160 bits. Algoritmos acima de SHA-2 [37], com 256 bits, podem ser considerados cifras fortes [38]. Na tabela 2.1 estão alguns modelos.

Tabela 2.1: Comparação dos tipos de funções de *hash*[39].

Name	Block size (bits)	Word size (bits)	Output size (bits)	Rounds	Year of the standard
MD4	512	32	128	48	1990
MD5	512	32	128	64	1992
SHA-1	512	32	160	80	1995
SHA-512	1024	64	512	80	2002

Além destas, podem ser consideradas outras técnicas úteis que serão explicadas adiante, como o *aHashing*, *pHashing*, *dHashing* e *wHashing*.

2.5.1 Técnicas de *hashing* perceptivas

Na seção anterior, foram apresentadas funções de *hash* criptográficas mais comuns e, apesar desses algoritmos terem sido vastamente estudados e aplicados, existem casos em que é necessário o uso de outro tipo de técnica de *fingerprinting*. Por exemplo, na comparação de imagens.

Devido à natureza das funções de *hashing* criptográficas, pequenas mudanças na entrada acarretam em saídas completamente diferentes. Quando se deseja comparar imagens, a única possibilidade utilizando esta implementação seria encontrar imagens exatamente iguais, que logicamente possuiriam a mesma *hash*.

No entanto, caso as imagens sejam minimamente diferentes, em um único pixel, seria interessante um algoritmo que possa interpretar essas pequenas diferenças e ainda calcular uma similar identidade para esse arquivo, de acordo com a natural percepção do ser humano. Estas técnicas, que serão apresentadas a seguir, produzem colisões de *hash* para entradas semelhantes e compõem uma classe de funções *hash* que podem ser utilizadas na comparação de *fingerprints* entre arquivos [40].

2.5.2 *aHash: average Hash*

O mais simples dentre os algoritmos perceptivos aqui apresentados, o *aHash* avalia se o valor de um determinado pixel é maior ou menor que a média dos demais. Para tanto, são observados os procedimentos descritos em [41]:

1. A imagem é inicialmente redimensionada para n por n pixels;
2. É então tomada a escala de cinza da figura, deixando apenas um canal de informação;
3. É calculada a intensidade média de todos os pixels;
4. Para cada pixel obtido no passo 2, é comparado seu valor com a média obtida em 3, armazenando o resultado em um *array* de n^2 bits;
5. Esse *array* binário é então convertido para base hexadecimal, apresentando a *hash* desejada.

2.5.3 *dHash: difference Hash*

Requer menos processamento que o *pHash*, porém é mais complexo que o *aHash*. São calculadas as diferenças entre as intensidades de pixels adjacentes e então é formando um

vetor booleano, de forma similar ao *aHash*. Entretanto, o redimensionamento da imagem é feito com uma coluna adicional, pois desta forma haverá n diferenças entre n+1 colunas. São observados procedimentos em [44]:

1. A imagem é inicialmente redimensionada para n+1 por n pixels;
2. É então tomada a escala de cinza da figura, deixando apenas um canal de informação;
3. São comparados pixels adjacentes: P[n] < P[n+1];
4. Esse array binário é então convertido para base hexadecimal, apresentando a *hash* desejada.

2.5.4 *pHash: perceptual Hash*

Requer maior poder computacional que o anterior, pois ao invés de usar o valor médio das intensidades como no *aHash*, é empregada a transformada discreta do cosseno, ou DCT e então calculada a média desses valores [42, 43]. São observados os seguintes procedimentos [40]:

1. A imagem é inicialmente redimensionada para n por n pixels;
2. É então tomada a escala de cinza da figura, deixando apenas um canal de informação;
3. Computa-se a DCT para a imagem, resultando em componentes de frequência e escalares;
4. Calcula-se a média para os componentes de menor frequência, excluindo o termo DC;
5. Para cada pixel obtido no passo 2, é comparado seu valor com a média obtida em 4, armazenando o resultado em um *array* de n2 bits;
6. Esse *array* binário é então convertido para base hexadecimal, apresentando a *hash* desejada.

2.5.5 *wHash: wavelet Hash*

Similar ao processo *pHash* que utiliza a DCT, a função *wHash* se vale de outra forma de representar frequências: a Transformada Discreta de *Wavelets*, DWT, da família Haar [45].

2.6 DISTÂNCIA DE *HAMMING*

A descoberta da similaridade entre diferentes imagens pode ser feita usando modelos matemáticos como a Distância de *Hamming* entre as *hashes* perceptivas das mesmas, resultando no número de posições entre duas entradas que façam com que elas difiram entre si. Sendo as *hashes* calculadas como vetores booleanos, o resultado será quantas alterações seriam necessárias para que ambos se tornem idênticos. O limiar considerado aceitável para tal pode ser otimizado de acordo com o tipo de entrada que será utilizada, dessa forma seriam agrupadas *hashes* de visualizações de binários de malwares de tipos parecidos. A vantagem deste método é que as identidades das imagens podem ser calculadas previamente e armazenados em um banco de dados. Posteriormente, é feito o cálculo acima sem a necessidade de empregar as funções de *hash* novamente, economizando-se tempo de processamento.

A biblioteca de funções científicas scipy[15] possui módulos e funções que calculam a distância de *hamming* normalizada entre duas entradas; isto é, o valor absoluto da diferença sobre o comprimento dos vetores unidimensionais que estão sendo tratados. Observa-se em Fig. 2.12 que a distância de *hamming* entre *arrays* de 1 dimensão u e v, é simplesmente a porção que apresenta componentes diferentes. Se u e v são vetores booleanos, distância de *hamming* é dada pela equação 2.2.

$$\frac{c_{01} + c_{10}}{n} \tag{2.2}$$

Em que cij é o número de ocorrências de u[k] = i e v[k] = j para k<n, sendo n o comprimento do vetor.

Parameters :	**u** : *(N,) array_like* Input array. **v** : *(N,) array_like* Input array.
Returns:	**hamming** : *double* The Hamming distance between vectors *u* and *v*.

Figura 2.12: Parâmetros da função de cálculo da distância de *hamming*.

2.7 TÉCNICAS DE COMPARAÇÃO DE IMAGENS

Adicionalmente, imagens podem ser comparadas por outros modos, como pelo Erro Quadrático Médio (MSE) ou Índice de Similaridade Estrutural (SSIM). MSE requer menos poder de processamento, mas também pode ser mais inacurado, já que é comparada a imagem como um todo, globalmente, ao invés do SSIM, que provê uma melhor estimativa de similaridade

[15] https://www.scipy.org/docs.html

ao comparar regiões de cada imagem. Ambos os algoritmos podem ser implementados em *Python* e estão referenciados na documentação da biblioteca de processamento de imagens scikit[16] e são descritos pelas equações 2.3 e 2.4.

$$MSE = \frac{1}{m \cdot n} \cdot \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} [I(i,j) - K(i,j)]^2 \tag{2.3}$$

$$SSIM(x,y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu^2 + \mu^2 + c_1)(\sigma_x^2\sigma_y^2 + c_2)} \tag{2.4}$$

2.8 VIRUS TOTAL

A plataforma VirusTotal [46], VT, fornece relatórios de escaneamento com base nos cerca de 60 programas antivírus (AV) que dispõe de acordo com o arquivo, *hash* ou site recebido do usuário. Sua interface está apresentada na Fig. 2.13. Ela foi utilizada para verificar se a amostra em análise já foi previamente detectada como maliciosa e qual a sua classificação de acordo com o AV escolhido. Como será discutido posteriormente na seção 3.2, elegeu-se o AV ESET-NOD32 como o padrão para a classificação dos tipos e rotulação das variantes das amostras.

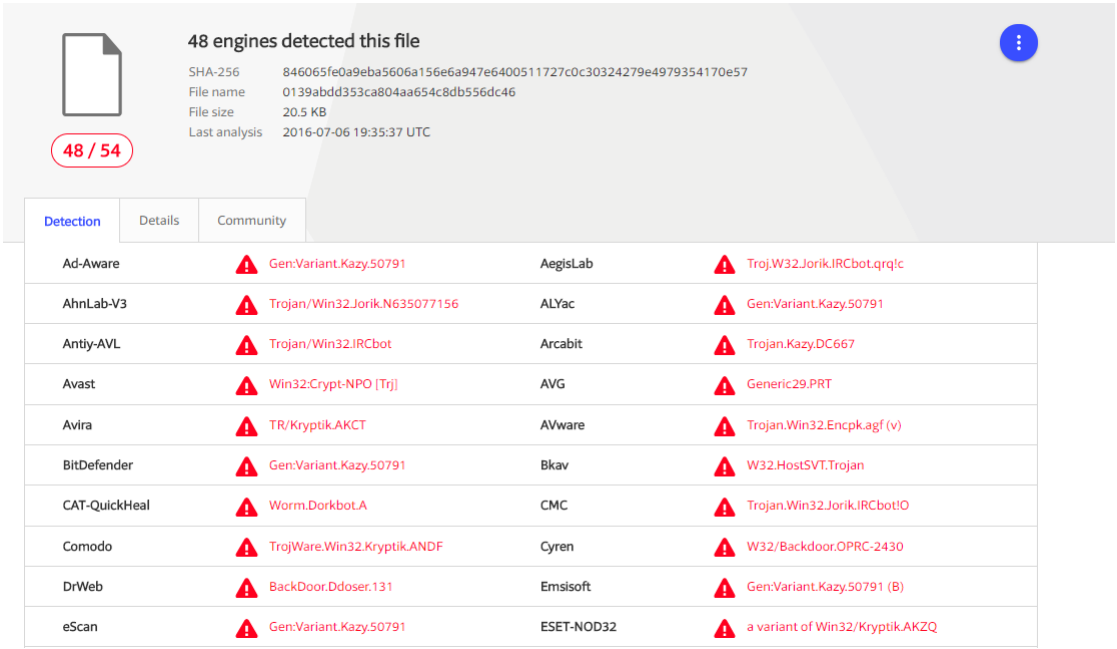

Figura 2.13: Interface da plataforma VirusTotal em resposta à uma ameaça identificada.

Por meio de sua API, é possível automatizar essa tarefa, utilizando scripts para realizar as consultas. As respostas são recebidas no formato JSON, cabendo ao usuário fazer o *parsing* em *strings* dos dados de interesse.

[16] http://scikit-image.org/docs

2.9 LINGUAGEM PYTHON

A linguagem de alto nível Python em sua versão 2.7 [47] foi empregada conjuntamente à IDE *Spyder* 3.2.8 no desenvolvimento dos algoritmos aplicados neste trabalho. Foram utilizados scripts que buscassem: as classificações de amostras na base de dados da plataforma VirusTotal via API pública do site; códigos que realizassem a leitura desses dados em formato csv (*comma separated values*); bibliotecas para processamento de imagem como o OpenCV [48]; leitura de arquivos; cálculo de *hashes*, classificação e detecção de *malwares*.

2.10 VIRTUALBOX

Com o software de virtualização VirtualBox, da Oracle, foi possível rodar um sistema operacional Windows XP contido em uma VM para melhor estudar e manipular os arquivos maliciosos, evitando pôr a máquina host sob risco de infecção ou perda de dados [49].

2.11 TRABALHOS RELACIONADOS

Além das técnicas apresentadas aqui, existem diversas outras que são interessantes de serem mencionadas neste trabalho. A visualização facilita o trabalho do analista do malware permitindo que de antemão ele tenha noção da estrutura da ameaça com a qual está lidando. Diferentes técnicas de visualização foram apresentadas para auxiliar a análise de *malwares*.

Em [50], foram empregados *Treemaps* e *Thread graphs* para visualizar o comportamento de *malwares*. Esses recursos proveem informações com relação a quantidade de chamadas API e comportamentos cronológicos da amostra.

Em [51], foi proposto um sistema que visualizasse sequências de chamadas do sistema significativas do registro e então construísse uma representação da estrutura dos binários de *malwares*. Esses dados, organizados em representação vetorial booleana, foram utilizados para traçar um mapa de similaridade entre as amostras selecionadas. Um sistema de visualização integrado com diversas técnicas gráficas foi apresentado em [52] por Conti.

Han, Kang e Im propuseram nos trabalhos [16] e [24] métodos para a comparação entre *malwares* utilizando, respectivamente, gráficos de entropia e representações visuais dos binários dos infecciosos. Estas representações foram construídas a partir dos *opcodes* obtidos após a realização do *disassembly* dos códigos maliciosos.

Em [53], Nandisha M. M. emprega as *hashes* perceptivas para combater a pirataria em aplicativos do sistema operacional Android. Para esse objetivo, o autor obtinha capturas de tela dos aplicativos e comparava as *hashes* com o banco de dados, possibilitando encontrar aplicativos com telas e recursos similares, indicando cópia não autorizada.

Em [54], foi encontrada a implementação do assunto tratado na seção 2.7, comparando uma imagem original e suas versões com contraste alterado e deformações com sobreposição de imagem. Em um primeiro momento, comparando a mesma imagem, o MSE foi 0 e o SSIM foi igual a 1, o que era esperado, pois se tratam de arquivos idênticos. A seguir, a comparação do arquivo original com aquele do contraste alterado também prosseguiu como previsto. Vale ressaltar que o MSE é proporcional à dissimilaridade, enquanto SSIM diminui. Mas, a terceira comparação evidenciou a discrepância entre as métricas apresentadas. O MSE diminuiu em relação à imagem com o contraste anterior, que claramente é mais próxima à original do que à sobreposta. Por outro lado, o Índice de Similaridade Estrutural diminuiu, alcançando o resultado esperado.

As aplicações são diversas. Seguindo linhas de pesquisa semelhantes, foram encontrados trabalhos na área de detecção de plágio, direitos autorais de imagens, propriedade intelectual, busca de imagens similares, filtros de pornografia e classificação de banco de dados.

Capítulo 3

MÉTODOS PROPOSTOS

Neste capítulo é apresentada a metodologia de análise proposta. São descritas as ferramentas, cenário e métodos empregados para a obtenção de resultados. Nas próximas seções serão detalhados os procedimentos acerca da obtenção das amostras e suas classificações com um antivírus escolhido, geração da representação visual de seus binários e comparação das mesmas.

3.1 OBTENÇÃO DO CONJUNTO DE AMOSTRAS

Para a construção de um método de comparação e classificação de *malwares*, é necessária uma base de binários. Esta base é composta, em sua maioria, por arquivos maliciosos e também por alguns arquivos benignos, nativos do sistema operacional e introduzidos para se avaliar a efetividade da classificação ou não em *malwares*.

3.1.1 Amostras de *malwares*

Em nothink.org [55], Matteo Cantoni reuniu diversos tipos de *malwares* oriundos de *honey pots*, links maliciosos e sites de pesquisa [56]. Realizou-se uma limpeza neste banco de amostras, eliminando arquivos HTML, de texto puro e executáveis não detectados por nenhum dos antivírus da plataforma VT. Um detalhamento sobre a verificação da maliciosidade de amostras será feito na seção seguinte. Utilizaram-se os exemplares remanescentes para a composição do banco de dados deste trabalho: são de arquivos executáveis para Windows e não foram desempacotados. Após estes passos, gerou-se um quantitativo de 2.175 amostras em 523 MB de dados em disco.

3.1.2 Arquivos não maliciosos

Aplicativos nativos do Windows, não infectados, foram coletados para integrar a base de comparação e testar a efetividade na detecção de *malwares*. Estes exemplares foram retirados no diretório *C:\Windows*. São estes os 7 arquivos: *calc.exe, cmd.exe, Defrag.exe, explorer.exe, mspaint.exe, notepad.exe e regedt32.exe*.

3.2 VERIFICAÇÃO DAS ASSINATURAS DAS AMOSTRAS NA PLATAFORMA VT

Aproveitando a documentação oferecida na plataforma[1], utilizou-se um código em *Python* para requisitar ao site os relatórios de escaneamento das amostras de *malware* obtidas na seção anterior. Por limitação da API pública, só puderam ser feitas 4 requisições por minuto, tomando 9 horas e 06 minutos para classificar as 2.175 amostras.

Escolheu-se como padrão para o presente trabalho a rotulação do Antivírus ESET NOD-32, o qual dispões de um glossário acessível com a descrição, classificação em família, variante e tipo de ameaça para as amostras na ferramenta Virus Radar, Fig. 3.1.

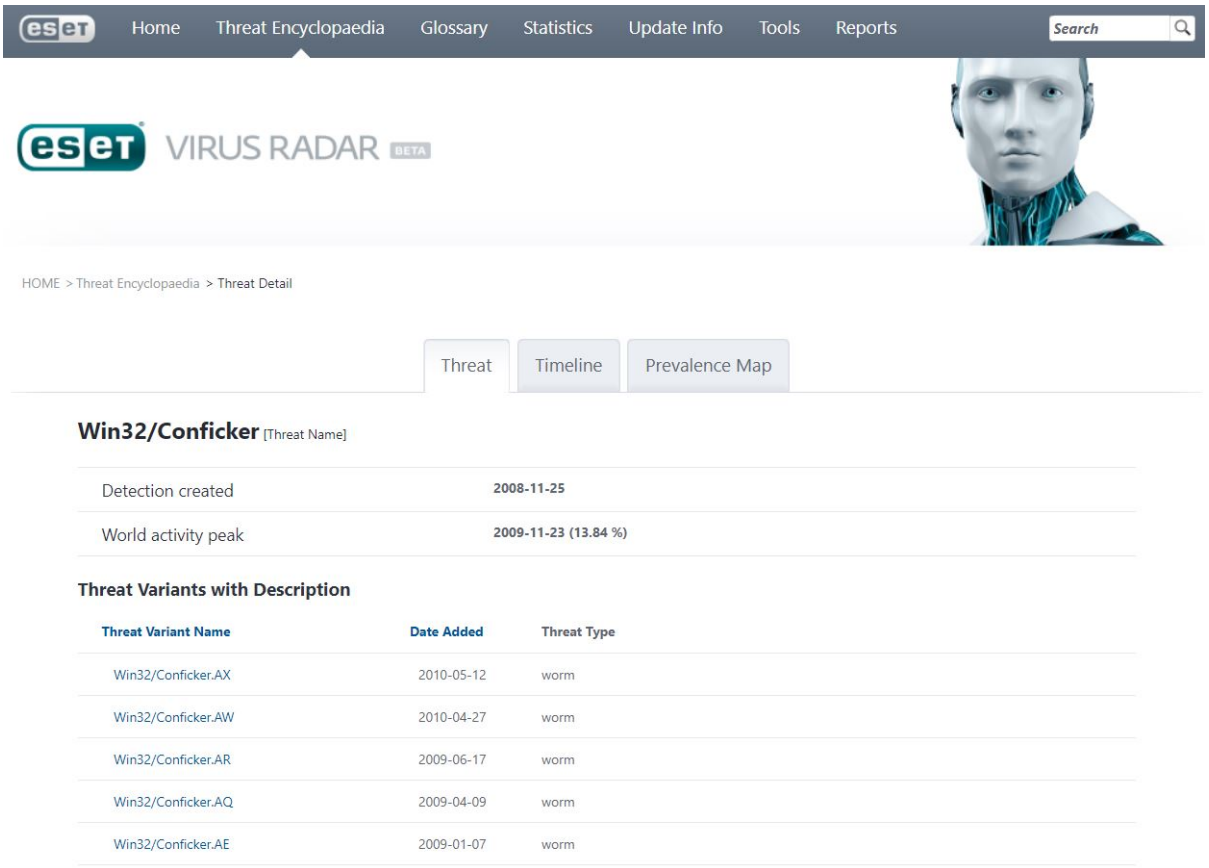

Figura 3.1: Tela do Virus Radar com informações sobre o *worm Conficker*

[1]https://www.virustotal.com/pt/

No código da Listagem 3.1, utilizou-se a chave pública do usuário, fornecida gratuitamente pelo VT, como parâmetro *apikey* para requisitar relatório de determinado *resource* — no caso uma *hash* MD5 de arquivo malicioso. No repositório [57], é indicado como melhor tratar os dados JSON fornecidos pelo VT.

Listagem 3.1: Código para requisição ao VT

```
1 import requests
2 params = {'apikey': '
    daf800c7724b6461cbd67d70e1b446bb10f8bbf610530120a5b3abe047e3a95d', '
    resource': '006afb5a73c4f77808a6a09bbe0515d1'}
3 headers = {
4   "Accept-Encoding": "gzip, deflate",
5   "User-Agent" : "gzip,  My Python requests library example client or
      username"
6   }
7 response = requests.get('https://www.virustotal.com/vtapi/v2/file/report'
    ,params=params,headers=headers)
8 json_response = response.json()
```

Após estes procedimentos, classificaram-se todas as 2.175 amostras componentes da massa de dados de acordo com o AV escolhido. Vale salientar que a questão da rotulação é muito particular de cada empresa, sendo que muitas vezes nem mesmo o tipo de ameaça é um consenso, tornando-se mais discrepante ainda se considerar família e variante.

Na Tabela 3.1, pode-se observar quantitativamente os tipos de amostra, em que o campo "*malwares* não detectados"corresponde àqueles cujas assinaturas foram tidas como limpas pelo AV ESET NOD-32.

Tabela 3.1: Composição da base de comparação: 2.182.

Tipos de amostra	**Amostras**
Malwares	2134
Malwares não detectados	41
Arquivos não maliciosos	7

Na Tabela 3.2, pode-se observar quantitativamente os tipos de *malware* entre aqueles previamente detectados, segundo o AV escolhido.

Tabela 3.2: Quantitativo dos tipos de ameaças detectada pelo AV padrão: 2134.

Tipos de ameaça	**Amostras**
Worm	2006
Trojan	102
Virus	26
Total	2134

Na Figura 3.2, pode-se observar quantitativamente as diferentes famílias de *malware*. Cerca de 80% das amostras são da família de *worm Conficker*, 7% *worm Autorun*, 3% *trojan Kryptic* e 2% *malwares* não detectados pelo AV.

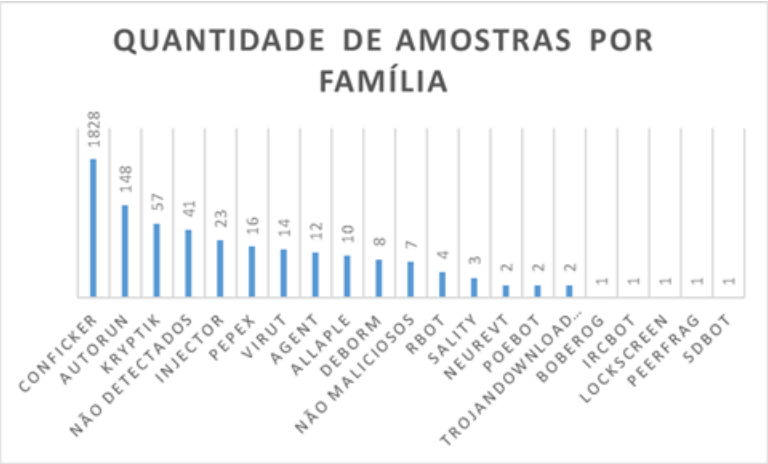

Figura 3.2: Distribuição de amostras por família de *malware*.

Com relação às variantes, a maior parte pertence à família *Conficker*, sendo que a mais comum é a AA, com 567 amostras (Fig. 3.3).

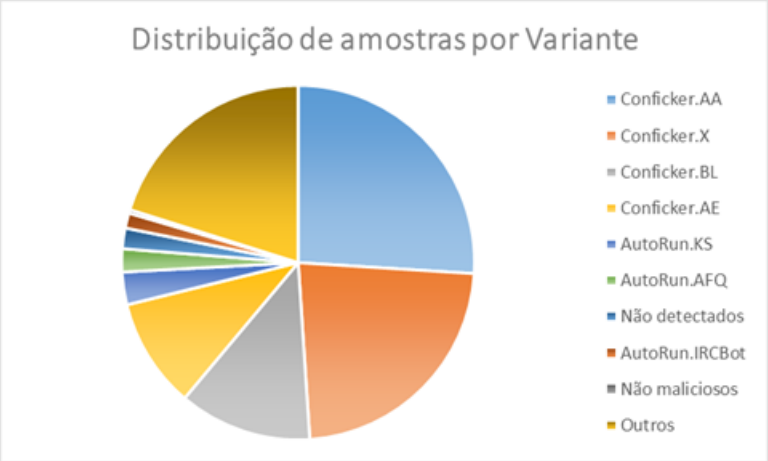

Figura 3.3: Distribuição de amostras por variante.

3.2.1 Falhas de detecção em AV conhecidos

Há casos, como o mostrado nas Figs. 3.4 e 3.5, que não foram detectados pelo AV escolhido, mas são ameaças identificadas por outros AV, e foram classificados como maliciosos pela metodologia aqui proposta. Na Figura 3.4, o arquivo de *hash* md5 "25bc5c80b91192d0738c5d2f47af06d9", que foi tido como limpo por 36 relatórios, apresenta exatamente as mesmas classificações entre diversos AV — tais quais o Avast, AVG, Ad-Aware e BitDefender, por exemplo — para a amostra "976ba6a95ee9bf23f6cff18b94d08aad" na Fig.3.5.

Isso demonstra a eficiência da proposta deste trabalho em avaliar as visualizações desses arquivos e chegar à mesma conclusão que os demais AV citados anteriormente, dentre outros, encontraram. É possível que o AV padrão escolhido tenha sido "enganado" com sucesso por esse código malicioso com mecanismos de ofuscamento e/ou empacotamento do seu código. Situação recorrente para outras amostras que não haviam sido identificadas pelo ESET NOD-32.

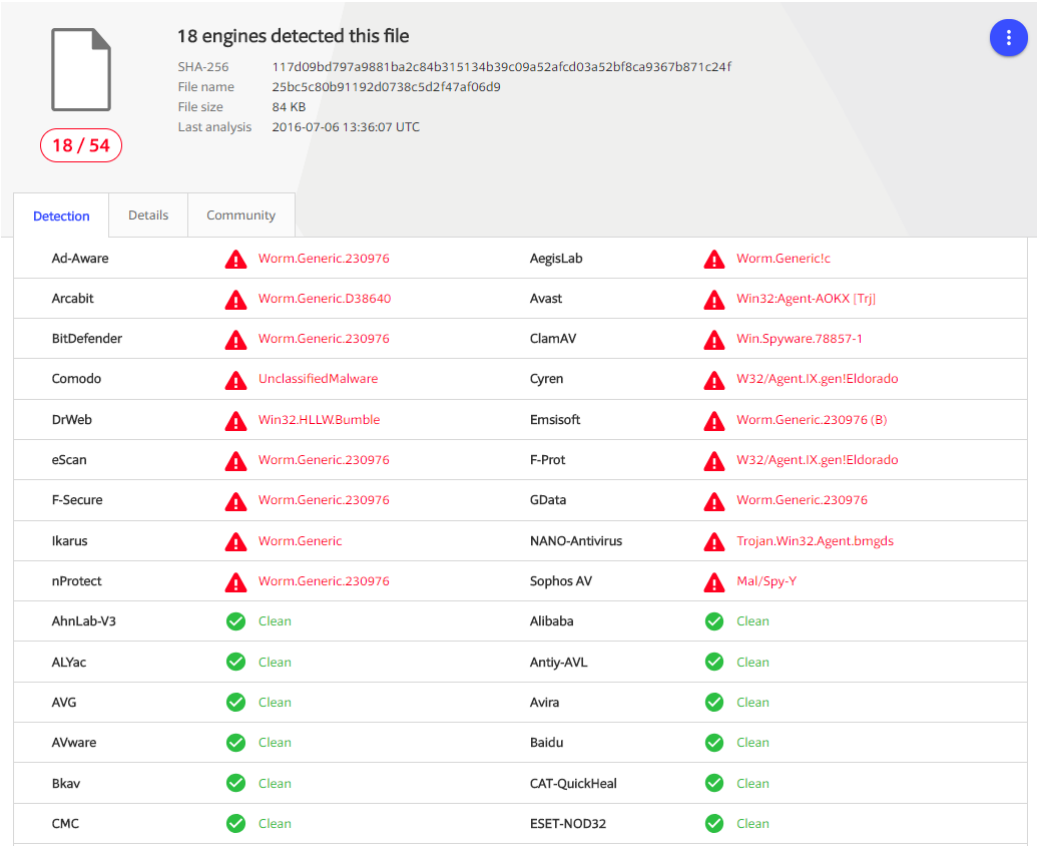

Figura 3.4: Relatório de escaneamento para arquivo 25bc5c80b91192d0738c5d2f47af06d9.

3.3 TÉCNICAS DE VISUALIZAÇÃO DO BINÁRIO DE *MALWARES*

Como fundamentado anteriormente, a técnica de representação visual do binário de *malwares* facilita bastante a tarefa de analistas. Rapidamente, pode-se detectar o tipo de arquivo, o nível de entropia e empacotamento, dando um rápido panorama. Na condução deste trabalho, priorizaram-se as análises pixel *byteclass* e *dotplot*. Para a realização das referidas verificações, utilizaram-se as ferramentes Binvis.io e BinVis. O funcionamento e o roteiro de ambas serão detalhados a seguir.

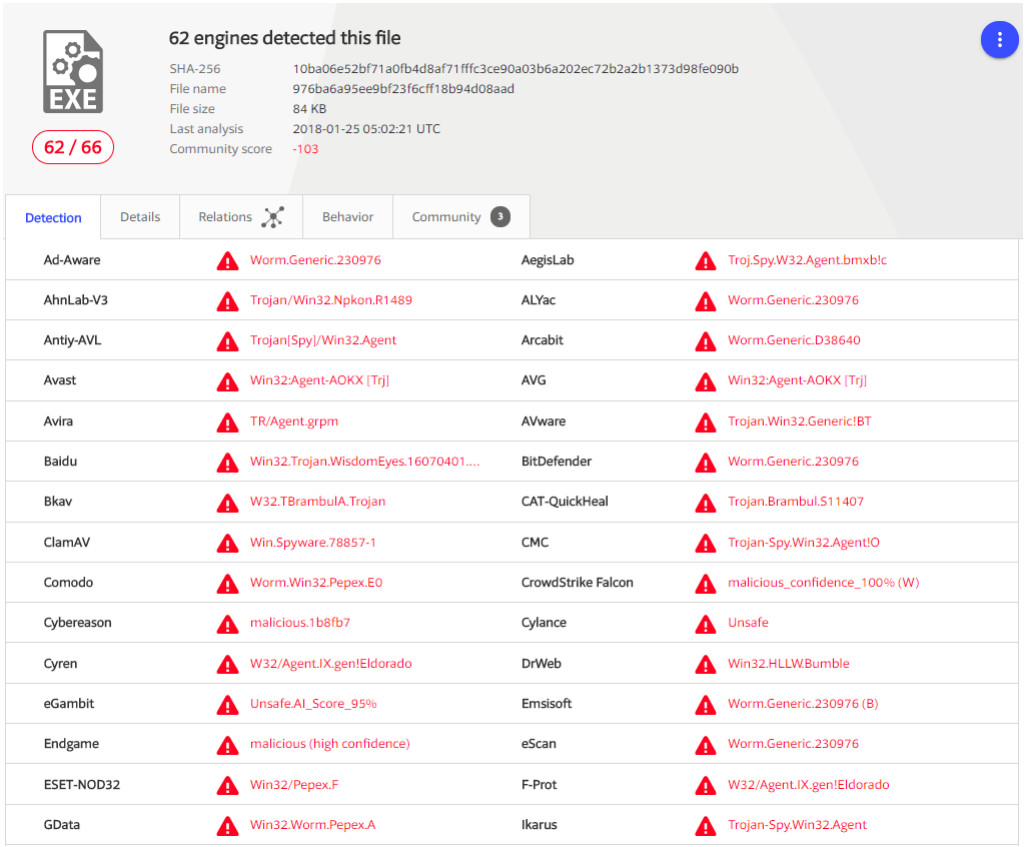

Figura 3.5: Relatório de escaneamento para o arquivo 976ba6a95ee9bf23f6cff18b94d08aad.

3.3.1 BinVis.io

O Binvis.io[2] é uma plataforma criada por Aldo Cortesi, empreendedor neozelandês da área de segurança. Seu trabalho foca na técnica pixel para a visualização dos binários de arquivos submetidos ao website. O objetivo é agilizar o reconhecimento de padrões e informações úteis em um determinado binário, conjugando a representação visual com um editor de código hexadecimal.

A interface da ferramenta é composta por uma página inicial, onde é realizada a submissão do arquivo para análise no botão *"open file"*, e uma página que apresenta os resultados. As respectivas telas são evidenciadas nas figuras 3.6 e 3.7.

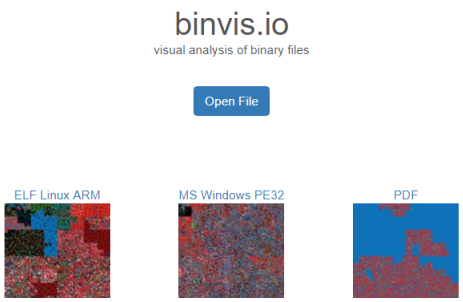

Figura 3.6: Tela inicial do binvis.io.

[2]http://binvis.io/

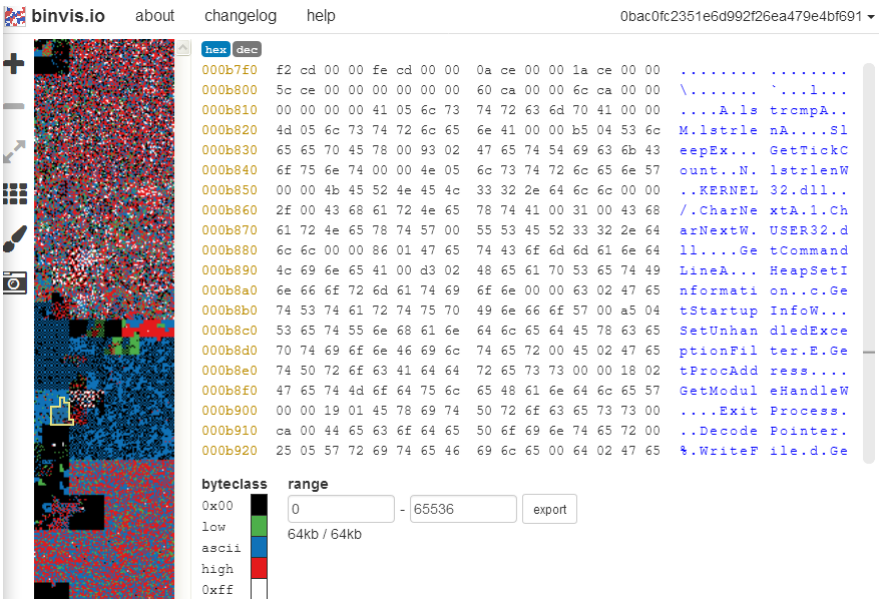

Figura 3.7: Apresentação de resultados da ferramenta para a amostra *trojan* 0bac0fc2351e6d992f26ea479e4bf691.

O Binvis.io divide os bytes de acordo com seus valores em quatro grupos, colorindo-os de formas diferentes. Possui duas formas de preenchimento da imagem, o *Cluster* e o Sequencial. Cada pixel representa um byte de informação. O modo *Cluster* utiliza a técnica de Curvas de Hilbert, na qual é realizada a tentativa de preservar a localidade da informação. Isto é, bytes que estejam próximos em uma mesma dimensão são organizados para que estejam sempre agrupados quando plotados bidimensionalmente. No modo sequencial, os pixels são desenhados sequencialmente, linha a linha. Estas opções estão representadas na Fig. 3.8.

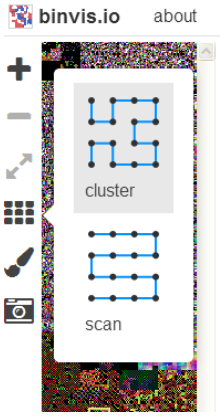

Figura 3.8: Tipos de curva oferecidas para o preenchimento da imagem.

A ferramenta ainda oferece diferentes possibilidades quanto à coloração dos pixels, gerando quatro resultados distintos ideais para cada tipo de análise. São elas: *Byteclass*, Magnitude, Detalhe e Entropia.

No modo *Byteclass*, os pixels são divididos em cinco grandes grupos que permitem uma rápida visão geral do arquivo: *low bytes* (são códigos de controle e meta-dados), ascii para o texto de representação visual, high bytes para outros códigos utilizados e duas categorias

especiais para a menor e a maior representação possível em um byte, 0x00 e 0xFF, respectivamente em hexadecimal. Estes valores costumam ser aplicados no preenchimento de seções vazias.

O Modo Magnitude pode revelar pequenos detalhes estruturais que ficam camuflados no esquema *Byteclass*, gerando 4 cores que dividem igualmente os valores possíveis de um byte. Os tons aplicados variam gradativamente de acordo com a categoria na qual o byte é classificado.

O Modo Detalhe se assemelha ao Magnitude, porém tenta aplicar cores mais discrepantes para uma diferente percepção.

Por último, há o Modo Entropia, o qual realiza o cálculo da Entropia de Shannon, obtendo a similaridade entre os bytes próximos. Um valor alto de entropia representa bytes próximos muito diferentes uns dos outros, o que pode representar uma seção ofuscada, empacotada, comprimida e criptografada. A entropia em áreas de texto costuma variar entre ordenada e média, sendo as áreas em rosa as completamente aleatórias. Na Figura 3.9 é possível observar o esquema de cores atribuído para cada modo de visualização dos dados.

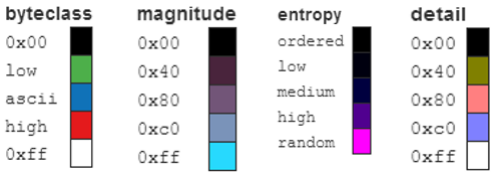

Figura 3.9: Legenda para os diferentes modos de visualização.

Na Figura 3.10, apresenta-se, para o modo *Byteclass*, as curvas de visualização sequencial e clusterizada, além do modo de entropia.

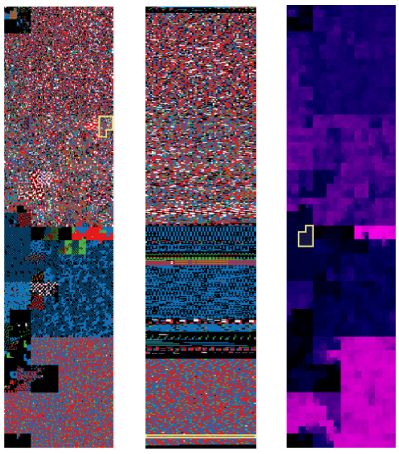

Figura 3.10: A mesma amostra nas curvas Cluster e Sequencial no modo *Byteclass*. Ao lado, o modo entropia também clusterizado.

Utilizou-se a ferramenta Binvis.io para gerar 2.182 imagens de dimensões 365x1460 pixels, para todas as amostras e arquivos benignos, totalizando um tamanho total de 245 MB em disco. As dimensões são definidas arbitrariamente pela ferramenta.

Na Figura 3.11, nota-se a dificuldade em distinguir a olho nu as visualizações para duas amostras *Conficker.AA*, comprovando a necessidade do uso de recurso computacional.

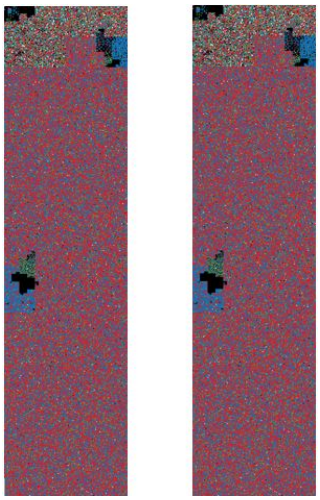

Figura 3.11: Visualizações Byteclass para duas amostras de mesma variante e família.

3.3.2 BinVis

BinVis é um software desenvolvido por Conti, Dean, Sinda e Sangster para o auxílio da engenharia reversa aplicada à segurança [52]. Foi levada em consideração a necessidade de se construir uma aplicação capaz de prover variadas formas de apresentação da informação, que incorporasse tanto análise textual quanto técnicas de visualização dos dados, combinando as boas práticas de editores hexadecimais com a possibilidade de percepções visuais inovadoras. Particularmente interessante, a análise *dotplot*, com seu cálculo de auto similaridade, foi escolhida para a geração das representações visuais dos binários das amostras utilizadas neste trabalho. Sua tela inicial está disposta em Fig. 3.12.

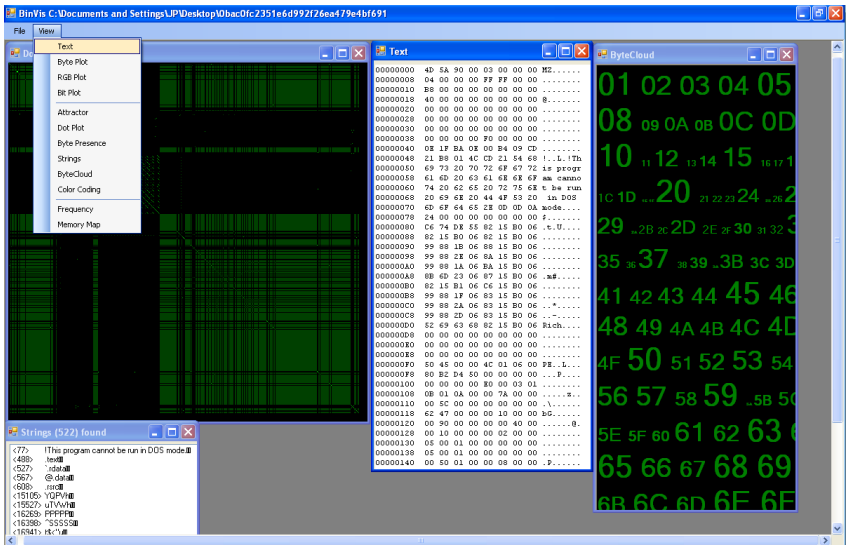

Figura 3.12: Tela inicial do aplicativo com diversas ferramentas em destaque.

Como observado na Figura 3.12, a aplicação possui inúmeras ferramentas para auxílio da análise estática de *malwares*. No menu *File* é realizado o carregamento da amostra. Na outra opção de menu, a *View*, se encontram as mais variadas ferramentas. Suas principais funções são:

- *Text*: mostra o conteúdo do arquivo em um visualizador de texto hexadecimal;
- *Byte Plot*: mapeia cada byte no arquivo em um pixel em uma janela;
- *RGB Plot*: como o nome já diz, um plot RGB com 3 bytes por pixel;
- *Bit Plot*: cada bit no arquivo é mapeado em uma janela, correspondendo a um pixel;
- *Attractor Plot*: plotagem baseada na teoria do caos;
- *dotplot*: cria em uma janela uma visualização baseada na sequência de bytes repetidos no arquivo;
- *Strings*: mostra as *strings* encontradas em um visualizador de texto;
- *Byte Cloud*: nuvem visual gerada a partir dos bytes que compõem o arquivo.

Com o Binvis, geraram-se 2.182 imagens de dimensões 510x509, armazenadas em 1,53 GB de espaço em disco. As dimensões são definidas arbitrariamente pela ferramenta.

Novamente, na Figura 3.13, nota-se a dificuldade em distinguir a olho nu as visualizações para duas amostras *Conficker.AA*, comprovando a necessidade do uso de recurso computacional.

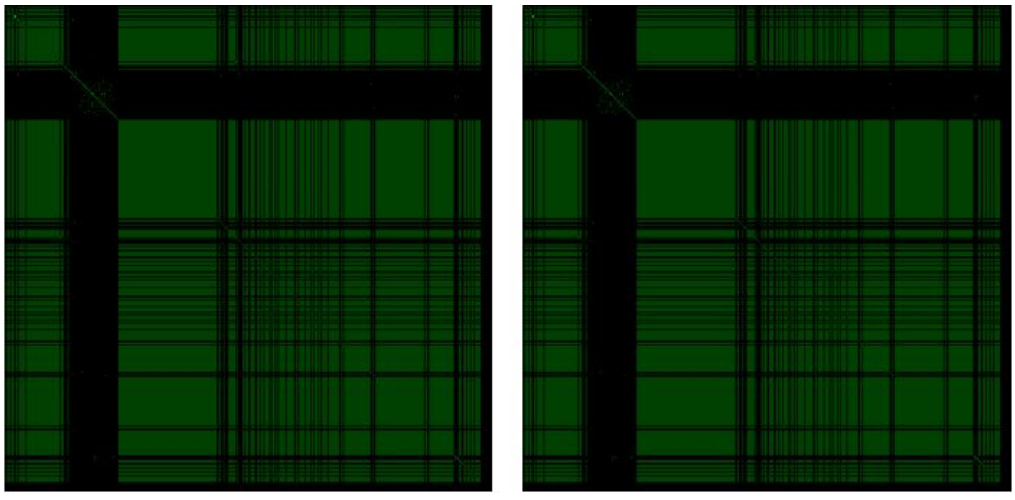

Figura 3.13: Representação visual *dotplot* de dois *worms* Conficker.AA.

Na Figura 3.14, é apresentada a representação visual para amostra idêntica a da Fig. 3.7.

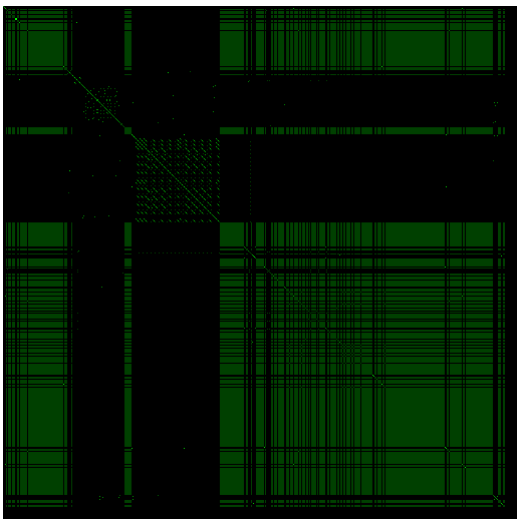

Figura 3.14: Representação *dotplot* para amostra *trojan* 0bac0fc2351e6d992f26ea479e4bf691.

3.4 IMPLEMENTAÇÃO

Foi feita uma limpeza inicial nas amostras coletadas, como foi descrito na metodologia deste trabalho (seção 1.3). Então, os algoritmos e bibliotecas fundamentados em 2.5 a 2.7 foram aplicados aos resultados obtidos com a classificação e visualização das amostras.

3.4.1 Ambiente de execução

Utilizou-se neste trabalho um laptop Acer Aspire 5, modelo A515-51-563W com as seguintes especificações:

- Processador Intel Core i5-7200U;
- 8 GB memória RAM DDR4 a 2133 MHz;
- SSD Samsung Evo 850 em *slot* M.2.

3.4.2 Código do algoritmo de comparação de *hashes*

Os algoritmos contém funções da bilioteca ImageHash [58], porém foram otimizadas e adaptadas para as necessidades deste trabalho.

O código mostrado na Listagem 5.1, contida no Apêndice B, foi desenvolvido com a seguinte linha de raciocínio:

1. É lido um arquivo contendo as classificações resultantes do processamento do VT e criado um dicionário com chaves sendo *hashes* e campos "família", "variante"e "tipo";

2. Em seguida, cada chave é buscada como um arquivo em um diretório contendo todas as visualizações;

3. Para cada imagem, são calculadas as *hashes* perceptivas apresentadas anteriormente e criados seus respectivos novos campos no dicionário anterior;

4. Todas as *hashes* de cada chave do dicionário têm seus respectivos valores comparados com todas as *hashes* demais chaves;

5. É feito um *loop* com 104.736 iterações (2.182*48) visando obter as menores diferenças para cada tipo de *hash*;

6. As chaves com que as inicialmente buscadas são mais próximas têm seus valores de "família", "variante"e "tipo"comparados com as originais;

7. Em caso de discrepância em algum desses campos, são incrementados contadores específicos, que visam determinar a quantidade de erros total para todas as 2.182 amostras;

8. Os resultados são escritos em um arquivo .txt.

3.4.3 Código do algoritmo de comparação de imagens

A biblioteca de processamento de imagem em *Python scikit-image* [59] fornece implementações para os algoritmos de similaridade de imagens discutidos no capítulo anterior. Dela, foram empregados os módulos *compare_ssim* e *compare_mse*.

O código apresentado na Listagem 5.2, contida no Apêndice B, foi desenvolvido de forma similar ao da subseção anterior, diferindo apenas quanto ao método empregado.

1. É lido um arquivo contendo as classificações resultantes do processamento do VT e criado um dicionário com chaves sendo *hashes* e campos "família", "variante"e "tipo";

2. Em seguida, cada chave é buscada como um arquivo em um diretório contendo todas as visualizações;

3. É feito um *loop* com 104.736 iterações (2.182*48) calculando o MSE e SSIM entre uma imagem e todas as demais do dicionário, visando obter as menores diferenças para cada imagem;

4. As chaves com que as inicialmente buscadas são mais próximas têm seus valores de "família", "variante"e "tipo"comparados com as originais;

5. Em caso de discrepância em algum desses campos, são incrementados contadores específicos, que visam determinar a quantidade de erros total para todas as 2.182 amostras;

6. Os resultados são escritos em um arquivo .txt.

3.4.4 Execução

Utilizando comparação de *hashes* para cada uma das duas visualizações, comparou-se a base de dados de *malwares* classificados consigo mesma para três diferentes resoluções escolhidas de modo a reduzir o tempo de processamento — 8 x 8, 100 x 100 e 300 x 300 pixels — em 4 técnicas de *hashing* perceptivas diferentes. O objetivo deste passo inicial foi apenas encontrar a configuração ótima que obtivesse maior taxa de acerto na classificação da família e tipo de *malware*; dado que essas classificações para a base comparada já eram conhecidas.

Em sequência, foram comparadas à base, as 41 amostras não detectadas mais os 7 arquivos limpos do Windows. Então calculou-se o percentual de acerto na detecção dos 41 *malwares* considerando que não houvesse falsos positivos.

3.4.5 Tratamento de dados

As saídas .txt resultantes dos códigos desenvolvidos foram tratadas no aplicativo Microsoft Excel, para melhor filtragem e manipulação dos dados.

Capítulo 4

RESULTADOS EXPERIMENTAIS

Neste capítulo são apresentados os principais resultados obtidos seguindo a metodologia disposta na introdução deste trabalho e detalhada na seção anterior. Os resultados serão apresentados com figuras, tabelas e terão apoio de material contido no Apêndice. Criaram-se diversas seções e subseções de acordo com o método empregado e resultado obtido. Buscou-se avaliar a eficácia das metodologidas em dois cenários: quanto à correta detecção de *malwares* não identificados previamente e quanto à classificação dos que haviam sido detectados em tipo e família.

4.1 PROVA DE CONCEITO

Primeiramente, como forma de testar o método proposto, realizou-se um teste como prova de conceito. Utilizando-se uma parte da massa de comparação que havia sido previamente rotulada com base no AV definido, aplicaram-se os algoritmos de comparação para as representações visuais das amostras. Foram alternados os métodos de *hashing* e de visualização, em busca da técnica que fosse mais acurada em relação à rotulação já conhecida. A comparação da base de 2.134 amostras consigo mesma, excluindo auto comparação de *malwares*, possibilitou a percepção do método *dotplot* em resolução média (100 x 100 pixels) como o mais preciso. Os resultados estão dispostos nas tabelas e subseções a seguir. Os valores nas tabelas referem-se às quantidades de amostras com correta classificação do dado em questão na coluna.

4.1.1 Observação quanto aos “erros”

Nas seções, subseções e tabelas seguintes, observa-se a ocorrência da terminologia "erro". Deve-se notar que este não necessariamente invalida o método aplicado. Apenas refere-se sobre o número de casos em que ocorreu uma má classificação em relação ao tipo da ameaça. Um *malware* classificado em um tipo errôneo ainda assim é detectado como um

arquivo infeccioso.

4.1.2 Visualização *byteclass*

Com a Tabela 4.1, podem-se observar os resultados ao redimensionar as imagens para uma resolução de apenas 8 x 8 pixels. O algoritmo wHash obteve 90% de acerto na rotulação do tipo de *malwares*.

Tabela 4.1: Resultados para aplicação dos algoritmos em imagens *Byteclass* com resolução mínima.

Algoritmo	**Variante**	**Família**	**Tipo**	**Erro**	**Erro percentual [%]**
aHash	604	1876	1876	258	12
dHash	712	1841	1841	293	13,73
pHash	752	1888	1888	246	11,52
wHash	761	1808	1937	197	9,23

Com a Tabela 4.2, podem-se observar os resultados ao redimensionar as imagens para uma resolução de 100 x 100 pixels. O algoritmo pHash obteve 91,5% de acerto na rotulação do tipo de *malwares*.

Tabela 4.2: Resultados para aplicação dos algoritmos em imagens *Byteclass* com resolução média.

Algoritmo	**Variante**	**Família**	**Tipo**	**Erro**	**Erro percentual [%]**
aHash	539	1402	1402	732	34,3
dHash	459	1657	1657	477	22,3
pHash	815	1954	1954	180	8,5
wHash	761	1808	1937	197	9,23

Com a Tabela 4.3, podem-se observar os resultados ao redimensionar as imagens para uma resolução de 300 x 300 pixels. Novamente wHash com 90% de acerto no tipo de *malware*.

Tabela 4.3: Resultados para aplicação dos algoritmos em imagens *Byteclass* com resolução máxima.

Algoritmo	**Variante**	**Família**	**Tipo**	**Erro**	**Erro percentual [%]**
aHash	354	897	897	1237	58
dHash	254	611	611	1523	71,3
pHash	721	1920	1920	214	10
wHash	761	1808	1937	197	9,23

4.1.3 Visualização *dotplot*

Com a Tabela 4.4, podem-se observar os resultados ao redimensionar as imagens para uma resolução de apenas 8 x 8 pixels. O algoritmo wHash obteve 92% de acerto na rotulação

do tipo de *malwares*.

Tabela 4.4: Resultados para aplicação dos algoritmos em imagens *dotplot* com resolução mínima.

Algoritmo	Variante	família	Tipo	Erro	Erro percentual [%]
aHash	258	1915	1915	198	9,27
dHash	450	1958	1958	155	7,26
pHash	325	1906	1906	207	9,7
wHash	460	1926	1967	146	6,8

Com a Tabela 4.5, podem-se observar os resultados ao redimensionar as imagens para uma resolução de 100 x 100 pixels. Os algoritmos pHash e aHash obtiveram 93,2% de acerto na rotulação do tipo de *malwares*.

Tabela 4.5: Resultados para aplicação dos algoritmos em imagens *Dotplot* com resolução média.

Algoritmo	Variante	Família	Tipo	Erro	Erro percentual [%]
aHash	522	1989	1989	124	5,8
dHash	517	1984	1984	129	6
pHash	551	1989	1989	124	5,8
wHash	460	1926	1967	146	6,8

Com a Tabela 4.6, podem-se observar os resultados ao redimensionar as imagens para uma resolução de 300 x 300 pixels. O algoritmo aHash obteve 93% de acerto na rotulação do tipo de *malwares*.

Tabela 4.6: Resultados para aplicação dos algoritmos em imagens *Dotplot* com resolução máxima.

Algoritmo	Variante	Família	Tipo	Erro	Erro percentual [%]
aHash	477	1985	1985	128	5,9
dHash	578	1983	1983	130	6
pHash	537	1981	1981	132	6,1
wHash	460	1926	1967	146	6,8

4.1.4 MSE

A comparação de imagens demorou muito mais que a de *hashes*, levando 17 horas no total para computar apenas 1/5 da quantidade de amostras. A taxa de acerto de variante foi de 53%, enquanto a de família ficou em 82%.

4.1.5 SSIM

O índice de similaridade estrutural não se mostrou uma técnica viável, demorando, em média, mais de 100 horas para finalizar a computação dos dados.

4.1.6 Taxa de acerto para Distâncias de *Hamming* menores

Os números informados na seção 4.1 referem-se a toda a base de *malwares* classificados e sem otimização quanto à distância de *hamming*. Por exemplo, com uma base de dados suficientemente grande, seria possível limitar a diferença máxima suportada para 1%.

Dessa forma, poderia haver um cenário como o da Fig. 4.1, em que foram selecionadas apenas amostras classificadas com pelo menos 99% de similaridade, resultando em 94% de acertos. A Tabela 5.1 com esses dados está contida no Apêndice "A"deste trabalho.

Na Figura 4.1, é possível notar a correlação entre a taxa de acertos e a similaridade das figuras. Quanto mais próximas, maior a chance de acerto na classificação da variante.

Figura 4.1: Correlação entre quantidade de acertos x *Hamming*.

4.2 DETECÇÃO DE *MALWARES*

As representações visuais de binários não detectados e de arquivos legítimos do Windows foram comparadas com a massa de amostras e dispostas nas tabelas a seguir, que tem como unidade a quantidade destes que foram detectados como *malwares*. Idealmente, todas as 41 amostras não detectadas deveriam ser classificadas como maliciosas e nenhum dos 7 arquivos benignos do Windows deveria ser detectado. Como dito na subseção 3.4.4 Execução, os parâmetros do código foram ajustados para que não houvesse falsos positivos.

A Tabela 4.7 exibe o melhor resultado, obtido com o modo de visualização *byteclass* em resolução máxima para o algoritmo aHash, com distância de Hamming de 29,1%.

Tabela 4.7: Melhor resultado obtido na detecção de arquivos maliciosos.

Categoria	**Detectados**	**Taxa de acerto [%]**
Amostras não classif.	36	87,6
Arquivos limpos	0	100

4.2.1 Visualização *byteclass*

Com a Tabela 4.8, podem-se observar os resultados ao redimensionar as imagens para uma resolução de apenas 8 x 8 pixels. O melhor resultado foi alcançado com o dHash e distância de Hamming de 35%.

Tabela 4.8: Quantidade de exemplares classificados como maliciosos em resolução mínima.

Categoria	**Detectados**	**Taxa de acerto [%]**
Amostras não classif.	31	75,6
Arquivos limpos	0	100

Com a Tabela 4.9, podem-se observar os resultados ao redimensionar as imagens para uma resolução de 100 x 100 pixels. O melhor resultado foi alcançado com o pHash e distância de Hamming de 47,7%.

Tabela 4.9: Quantidade de exemplares classificados como maliciosos em resolução média.

Categoria	**Detectados**	**Taxa de acerto [%]**
Amostras não classif.	35	85,3
Arquivos limpos	0	100

A Tabela 4.10 exibe o melhor resultado, obtido com o modo de visualização *byteclass* em resolução máxima para o algoritmo aHash, com distância de Hamming de 29,1%.

Tabela 4.10: Quantidade de exemplares classificados como maliciosos em resolução máxima.

Categoria	**Detectados**	**Taxa de acerto [%]**
Amostras não classif.	36	87,6
Arquivos limpos	0	100

4.2.2 Visualização *dotplot*

Com a Tabela 4.11, podem-se observar os resultados ao redimensionar as imagens para uma resolução de apenas 8 x 8 pixels. O melhor resultado foi alcançado com o wHash e distância de Hamming de 21%.

Tabela 4.11: Quantidade de exemplares classificados como maliciosos em resolução mínima.

Categoria	**Detectados**	**Taxa de acerto [%]**
Amostras não classif.	6	14,63
Arquivos limpos	0	100

Com a Tabela 4.12, podem-se observar os resultados ao redimensionar as imagens para uma resolução de 100 x 100 pixels. O melhor resultado foi alcançado com o pHash e distância de Hamming de 48%.

Tabela 4.12: Quantidade de exemplares classificados como maliciosos em resolução média.

Categoria	**Detectados**	**Taxa de acerto [%]**
Amostras não classif.	6	14,63
Arquivos limpos	0	100

Com a Tabela 4.15, podem-se observar os resultados ao redimensionar as imagens para uma resolução de 300 x 300 pixels. O melhor resultado foi alcançado com o dHash e distância de Hamming de 19%.

Tabela 4.13: Quantidade de exemplares classificados como maliciosos em resolução máxima.

Categoria	**Detectados**	**Taxa de acerto [%]**
Amostras não classif.	34	82,9
Arquivos limpos	0	100

4.2.3 MSE

Com a Tabela 4.15, podem-se observar os resultados ao redimensionar as imagens para uma resolução de 300 x 300 pixels. O melhor resultado foi alcançado com o dHash e distância de Hamming de 19%.

Tabela 4.14: Quantidade de exemplares classificados como maliciosos em resolução máxima.

Categoria	**Detectados**	**Taxa de acerto [%]**
Amostras não classif.	28	68,2
Arquivos limpos	0	100

4.2.4 SSIM

Com a Tabela 4.15, podem-se observar os resultados ao redimensionar as imagens para uma resolução de 300 x 300 pixels. O melhor resultado foi alcançado com o dHash e distância de Hamming de 19%.

Tabela 4.15: Quantidade de exemplares classificados como maliciosos em resolução máxima.

Categoria	Detectados	Taxa de acerto [%]
Amostras não classif.	34	82,9
Arquivos limpos	0	100

4.2.5 Ajuste para o melhor resultado

Nesta subseção, o algoritmo aplicado para o melhor resultado encontrado na seção 4.2 foi otimizado de forma que todas as ameaças pudessem ser detectadas. Isto foi feito partindo do ponto de vista de que é mais interessante alarmar arquivos limpos do que ignorar ameaças reais.

Os resultados podem ser vistos na Tab. 4.16. Com o novo ajuste, nota-se a ocorrência de 2 falsos positivos para a visualização *byteclass* em resolução máxima com o método dHash e uma distância de Hamming de 46,64%.

Tabela 4.16: Quantidade de exemplares classificados como maliciosos em resolução máxima.

Categoria	Detectados	Taxa de acerto [%]
Amostras não classif.	41	100
Arquivos limpos	2	81,5

4.3 Tempos de processamento

Nas Tabelas 4.17 e 4.18 são apresentadas as performances no caso da prova de conceito e das 41 amostras sem classificação somadas aos arquivos legítimos. As diferentes técnicas podem ter suas performances comparadas para cada caso. Os tempos de execução estão consistentes com a teoria apresentada, bem como a relação entre a complexidade dos mesmos e eficácia no acerto da classificação.

Tabela 4.17: Comparativo dos tempos de execução dos algoritmos para a base de comparação. *valores estimados por limitação do *hardware*.

Técnica	Res. mínima [s]	Res. média [s]	Res. máxima [s]
byteclass	358	766	4102
dotplot	371	731	4123
byteclass mse	-	-	61200*
byteclass ssim	-	-	457000*

Tabela 4.18: Comparativo dos tempos de execução dos algoritmos para as amostras não detectadas somadas aos arquivos limpos.

Cenário	Res. mínima [s]	Res. média [s]	Res. máxima [s]
byteclass	97	132	391
dotplot	102	124	363
byteclass mse	-	926	-
byteclass ssim	-	1031	-

4.4 Ganhos temporais em comparação com análises dinâmica e estática

Tomando como referência os valores do tempo total de processamento, 4102 segundos (Tab. 4.17), ao utilizar a técnica *ahash* para imagens *byteclass* máximas (300 x 300 pixels), é possível afirmar que a classificação de um arquivo como malicioso ou não é agilizada enormemente.

Métodos comuns para essa detecção, como a análise dinâmica ou estática, requerem muitas horas ou até dias de análise por amostra. Isto significa que é possível obter uma redução do tempo necessário de no mínimo 95,2%, considerando vinte e quatro horas de trabalho para analisar uma amostra com as técnicas tradicionais.

4.5 Síntese dos resultados

As seções anteriores possibilitaram observar diversos aspectos sobre o tema.

Quanto à resolução, nota-se que a eficácia dos métodos propostos não é diretamente proporcional às dimensões da imagem, enquanto o tempo de processamento aumenta em até 1045%, considerando a diferença entre resolução mínima para a máxima na visualização *byteclass*.

Quanto ao modo de visualização, em geral o *dotplot* apresentou maior acerto na determinação da família e tipo de ameaça. Por outro lado, a representação *byteclass* foi mais precisa para a classificação da variante de *malwares*.

Quanto ao método de comparação, o pHash obteve melhores resultados para *byteclass* enquanto o aHash errou menos o tipo de ameaça para imagens *dotplot*.

Quanto à correta classificação, houve um acerto de até 94,2% na correta classificação do tipo de *malware* para aHash e pHash em resolução média para *dotplot*.

Quanto à correta detecção, mostrou-se a versatilidade do método proposto, visto que o mesmo pode ser otimizado de forma a detectar a totalidade dos 41 não identificados pelo AV escolhido apresentando apenas 2 falsos positivos.

Capítulo 5

CONCLUSÕES

A Segurança da Informação vive um grande desafio na atualidade. O surgimento de milhares de novas ameaças todos os dias, somado ao fato do interesse cada vez maior por esse mercado dos ataques, faz com que a descoberta de novas técnicas de reconhecimento e classificação de *malwares*, e melhoria das atuais, sejam imperativas.

Através de representações visuais dos binários, torna-se mais simples a análise destas ameaças por parte dos analistas de segurança. Conforme o tipo e enfoque da representação, são variadas possibilidades para a inspeção de arquivos. Utilizando-se de algoritmos para a comparação de imagens, é possível desenvolver novos caminhos para a classificação de amostras infecciosas apenas tratando a visualização de seus binários.

De posse destes novos métodos, diferentes testes foram impostos à metodologia. O tipo de ameaças da base de amostras foi corretamente classificado em 94,2% das vezes para 2.134 *malwares*, previamente detectados.

Como foi observado nas seções anteriores, a técnica mais eficaz depende do objetivo do avaliador. A representação *byteclass* mostrou-se melhor para classificação das variantes, enquanto *dot plot* na detecção do tipo e família do *malware*. Os dados obtidos contrariaram a expectativa inicial de uma acurácia diretamente relacionada à resolução das imagens tratadas. O processo de redimensioná-las, visando simplificar os cálculos, entrega bons resultados.

Em seguida, 7 arquivos legítimos foram recolhidos do diretório do Windows para serem testados conjuntamente com 41 amostras que não haviam sido classificadas pelo AV de escolha, o ESET NOD-32. Sem alterar os métodos propostos, foi alcançada uma eficácia de 87,6% na detecção dos arquivos como maliciosos enquanto corretamente não detectando arquivos limpos do Windows. Com ajustes, todas as 41 amostras foram detectadas como maliciosas enquanto houve apenas 2 falsos positivos para os arquivos limpos.

Além disto, como observado na seção 3.2.1, foi possível observar o acerto do método proposto em reconhecer ameaças enquanto diversos AV conhecidos tratavam estas amostras como não infectadas.

Foi demonstrado na prática como a técnica de *fingerprinting* de arquivos facilita a sua comparação, seja na identificação de arquivos idênticos ou na forma inovadora proposta por este trabalho para a verificação de similaridade.

Os algoritmos de comparação de imagens tomam muito mais tempo e processamento da máquina utilizada. Enquanto o código desenvolvido com todas as funções de *hash* perceptivo demorou cerca de 10 minutos para processar e classificar toda a informação, a comparação de imagens via similaridade estrutural e erro quadrático médio demorou horas.

Diferentes procedimentos e ideias foram continuamente atualizados em busca de melhores resultados. Pode ser possível empregar um sistema híbrido com mais de um mecanismo de classificação: um menos custoso, computacionalmente, para fazer uma filtragem inicial seguido de uma segunda fase mais complexa, por exemplo.

Portanto, é possível concluir que os resultados obtidos foram bastante satisfatórios. Com uma base de dados maior, os resultados devem se tornar ainda mais precisos. Ainda que não dispense outros tipos de análises estática e dinâmicas, é possível incrementar esta técnica aqui apresentada para servir de complemento na análise de arquivos infectados.

5.1 TRABALHOS FUTUROS

Dando continuidade a este trabalho, podem ser aproveitadas as técnicas e algoritmos aqui discutidos para a elaboração de um aplicativo que automatize os procedimentos apresentados, gerando uma ferramenta totalmente autônoma para a detecção e classificação de malwares. No intuito de aperfeiçoar este projeto, notaram-se alguns pontos de melhoria:

- A utilização de uma GPU para execução dos códigos proveria ganho comprovado de performance em relação ao uso de CPU para processamento. Em [61] esse ganho foi de 100x no tempo de execução;
- Combinação das funções de *hash* e das visualizações de modo a extrair o melhor resultado de cada;
- Melhorar a classificação quanto as variantes;
- Introduzir aprendizado de máquina para que se possa classificar amostras automaticamente;
- Integração das ferramentas BinVis e Binvis.io ou desenvolvimento de meios próprios para visualização dos binários;
- Desenvolver uma solução em nuvem completa para a detecção e classificação de amostras.

Referências Bibliográficas

[1] TRINIUS, P.; HOLTZ, T.; GOBEL, J.; FREILING, F. C. Visual analysis of malware behavior using treemaps and thread graphs. In: **2009 6th International Workshop on Visualization for Cyber Security (VizSec 2009)**. IEEE, 2009. p. 33-38.

[2] NOKIA. *Malware Report*. Acessado em agosto de 2018. Disponível em: <https://www.nokia.com/en_int/news/releases/2017/03/27/nokia-malware-report-reveals-new-all-time-high-in-mobile-device-infections-and-major-iot-device-security-vulnerabilities>.

[3] GARTNER.*Wordwide Information Security Spending*. Acessado em agosto de 2018. Disponível em: <https://www.gartner.com/newsroom/id/3784965>

[4] BUSINESS INSIDER. *Cyber Security Importance*. Acessado em agosto de 2018. Disponível em: <http://www.businessinsider.com/cybersecurity-report-threats-and-opportunities-2016-3>

[5] AVTEST. *Malware Statistics*. Acessado em agosto de 2018. Disponível em: <https://www.avtest.org/en/statistics/malware/>.

[6] INFOMIT. *As the worm Turns: the Stuxnet Legacy*. Acessado em agosto de 2018. Disponível em: <http://www.informit.com/articles/article.aspx?p=1686289>.

[7] SIKORSKI, Michael; HONIG, Andrew. **Practical malware analysis: the hands-on guide to dissecting malicious software**. no starch press, 2012.

[8] MCAFEE. *Economic Impact of Cybercrime*. Acessado em agosto de 2018. Disponível em: <https://csis-prod.s3.amazonaws.com/s3fs-public/publication/economic-impact-cybercrime.pdf>

[9] SANS INSTITUTE. *Malware Analisys Introduction*. Acessado em agosto de 2018. Disponível em: < hrefhttps://www.sans.org/reading-room/whitepapers/malicious/malware-analysis-introduction-2103%20>

[10] VENTURE BEAT. *Why Security Vendors can't keep up with malware authors*. Acessado em agosto de 2018. Disponível em: <https://venturebeat.com/2011/08/04/why-security-vendors-cant-keep-up-with-malware-authors-and-what-to-do-about-it/>

[11] MALWARE FOX. *Malware types*. Acessado em agosto de 2018. Disponível em: <https://www.malwarefox.com/malware-types/>

[12] LAST LINE. *Malware types and classifications*. Acessado em agosto de 2018. Disponível em: <https://www.lastline.com/blog/malware-types-and-classifications/>

[13] SC MAGAZINE UK. *FBI says ransomware soon becoming a billion dolar business*. Acessado em agosto de 2018. Disponível em: <https://www.scmagazineuk.com/fbi-says-ransomware-soon- becoming-a-billion-dollar-business/article/630615/>

[14] INCAPSULA. *Botnet DDoS attacks*. Acessado em agosto de 2018. Disponível em: <https://www.incapsula.com/ddos/botnet-ddos.html>

[15] MALIN, Cameron H.; CASEY, Eoghan; AQUILINA, James M. **Malware Forensics Field Guide for Windows Systems: Digital Forensics Field Guides**. Elsevier, 2012.

[16] HAN, KyoungSoo; KANG, BooJoong; IM, Eul Gyu. Malware analysis using visualized image matrices. **The Scientific World Journal**, v. 2014, 2014.

[17] LEITE, LINDEBERG P. **Agrupamento de malware por comportamento de execução usando lógica fuzzy**. Universidade de Brasília, Distrito Federal, 2016.

[18] NTT SECURITY. *Hunting malware with memory analysis*. Acessado em agosto de 2018. Disponível em: <https://technical.nttsecurity.com/post/102egyy/hunting-malware-with-memory-analysis>

[19] MODY, Samir; MUTTIK, Igor; FERRIE, Peter. Standards and policies on packer use. In: **Virus Bulletin Conference**. 2010. p. 372-280.

[20] HAHN, Katja. Robust static analysis of portable executable malware. **Mater Thesis, HTWK Leipzig**, 2014.

[21] AYCOCK, John. **Computer viruses and malware**. Springer Science & Business Media, 2006.

[22] SOUZA, V. H. de. **Técnicas e Ferramentas de Análise Visual de *malwares***, Universidade de Brasília, Distrito Federal, 2016.

[23] TRINIUS, P., HOLTZ, T., GOBEL, J., FREILING, F. Visual analysis of malware behavior using treemaps and thread graphs. In: **2009 6th International Workshop on Visualization for Cyber Security (VizSec 2009)**. IEEE, 2009. p. 33-38.

[24] HAN, Kyoung Soo et al. Malware analysis using visualized images and entropy graphs. **International Journal of Information Security**, v. 14, n. 1, p. 1-14, 2015.

[25] GOODALL, John R.; CONTI, Gregory; MA, Kwan-Liu (Ed.). **VizSEC 2007: Proceedings of the Workshop on Visualization for Computer Security**. Springer Science & Business Media, 2008.

[26] KEIM, D. **Pixel-oriented Visualization Techniques for Exploring Very Large Databases**. Universidade de Munique. Munique. Acessado em agosto de 2018. Disponível em: <https://bib.dbvis.de/uploadedFiles/189.pdf>

[27] GRÉGIO, A. R. A.; FILHO, B. P. de C.; MONTES, A.; SANTOS, R. **Técnicas de visualização de dados aplicadas à segurança da informação**. Campinas, São Paulo, 2009

[28] KOHONEN, Teuvo et al. Engineering applications of the self-organizing map. **Proceedings of the IEEE**, v. 84, n. 10, p. 1358-1384, 1996.

[29] BÉLANGER, D. *Visualizing Patterns with Dotplots*. McGill University, Montreal. Acessado em agosto de 2018. Disponível em: <http://www.sable.mcgill.ca/dbelan2/courses/763/pres1/presentation.pdf>

[30] CONTI, Gregory et al. Visual reverse engineering of binary and data files. In: **Visualization for Computer Security**. Springer, Berlin, Heidelberg, 2008. p. 1-17.

[31] BRILLOUIN, Leon. **Science and information theory**. Courier Corporation, 2013.

[32] SHANNON, Claude Elwood. A Mathematical Theory of Communication. A**CM SIGMOBILE mobile computing and communications review**, v. 5, n. 1, p. 3-55, 2001.

[33] GRAVES, Russell Edward. **High performance password cracking by implementing rainbow tables on nVidia graphics cards** (IseCrack). 2008.

[34] TECHNOLOGY REVIEW. *Fingerprinting your files*. Acessado em agosto de 2018. Disponível em: <https://www.technologyreview.com/s/402961/fingerprinting-your-files/>

[35] IETF ORG. *The MD5 Message-Digest Algorithm*. Acessado em agosto de 2018. Disponível em: <https://tools.ietf.org/html/rfc1321>

[36] IETF ORG. *Us Secure Hash Algorithm 1 (SHA1)*. Acessado em agosto de 2018. Disponível em: <http://tools.ietf.org/html/3174>

[37] IETF ORG. *A 224-bit One-way Hash Function: SHA-224*. Acessado em agosto de 2018. Disponível em: <http://tools.ietf.org/html/3874>

[38] MIRONOV, I. Hash functions: **Theory, attacks and applications**. Nov. 2005. Acessado em agosto de 2018. Disponível em: <https://www.microsoft.com/en-us/research/wp-content/uploads/2005/11/hash_survey.pdf>

[39] National Institute of Standards and Technology (NIST), FIPS Publication 180-3: **Secure Hash Standard**, October 2008.

[40] CHAMOSO, Pablo et al. A hash based image matching algorithm for social networks. In: **International Conference on Practical Applications of Agents and Multi-Agent Systems**. Springer, Cham, 2017. p. 183-190.

[41] WÅLARÖ, Johannes. **Detecting visual plagiarism with perception hashing**. 2015.

[42] HACKER FACTOR BLOG. *Looks Like That*. Acessado em agosto de 2018. Disponível em: <http://www.hackerfactor.com/blog/?/archives/529-Kind-of-Like-That.html>

[43] NIXON, Mark S.; AGUADO, Alberto S. **Feature extraction & image processing for computer vision**. Academic Press, 2012.

[44] HACKER FACTOR BLOG. *Looks Like it*. Acessado em agosto de 2018. Disponível em: <http://www.hackerfactor.com/blog/index.php?/archives/432-Looks-LikeIt.html>

[45] BARBON, S. *Transformada Discreta Wavelet DWT*. Universidade Estadual de Londrina, Londrina. Acessado em agosto de 2018. Disponível em: <http://www.barbon.com.br/wp-content/uploads/2013/02/PDS_Aula4.pdf>

[46] VIRUS TOTAL. Acessado em agosto de 2018. Disponível em: <https://www.virustotal.com/pt/>

[47] PYTHON SOFTWARE FOUNDATION. *Python 2.7.15 documentation*. Acessado em agosto de 2018. Disponível em:<https://docs.python.org/2.7/>

[48] OPENCV. *OpenCV and Python tutorials*. Acessado em agosto de 2018. Disponível em: <https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_tutorials.html>

[49] VIRTUALBOX. *VirtulBox Documentation*. Acessado em agosto de 2018. Disponível em: <https://www.virtualbox.org/wiki/Documentation>

[50] MALIN, Cameron H.; CASEY, Eoghan; AQUILINA, James M. **Malware Forensics Field Guide for Windows Systems: Digital Forensics Field Guides**. Elsevier, 2012. p. 442-445

[51] SAXE, Josh; MENTIS, David; GREAMO, Chris. Visualization of shared system call sequence relationships in large malware corpora. In: **Proceedings of the ninth international symposium on visualization for cyber security**. ACM, 2012. p. 33-40.

[52] GOOGLE CODE. *Welcome to the BinVis Project*. Acessado em agosto de 2018. Disponível em: <https://code.google.com/archive/p/binvis/>

[53] NADISHA, M.; DESHPANDE, S. **Piracy Detection App of Android Aplications**. Visveswaraya Technological University. Belagavi. Acessado em agosto de 2018. Disponível em: <https://pdfs.semanticscholar.org/bcc7/59d00e9970a6ad63e5210fd07396bd29bc0d.pdf>

[54] SCIKIT-IMAGE. *MSE and SSIM*. Acessado em Novembro de 2017. Disponível em:<http://scikit-image.org/docs/dev/api/skimage.measure.html#structural-similarity>

[55] NOTHINK. *Malware Archives*. Acessado em agosto de 2018. Disponível em: <http://www.nothink.org/honeypots/malware-archives/>

[56] ZELTSER. *How to obtain malware samples*. Acessado em agosto de 2018. Disponível em: <https://zeltser.com/malware-sample-sources/>

[57] GITHUB SHILPESH TRIVEDI. *Multiple Hash Scan*. Acessado em agosto de 2018. Disponível em: <https://github.com/ShilpeshTrivedi/Multiple_Hash_Scan>

[58] GITHUB JOHANNES BUCHNER. *Image Hash*. Acessado em agosto de 2018. Disponível em: <https://github.com/JohannesBuchner/imagehash>

[59] WEERAMAN. *How to put GPU to good use with python*. Acessado em agosto de 2018. Disponível em:<https://weeraman.com/put-that-gpu-to-good-use-with-python-e5a437168c01>

APÊNDICE A

Tabela 5.1: Amostras a até 1% de dissimilaridade

md5_1	classificacao_1	md5_2	classificacao_2	hamming	Variante Igual
04732340b88bde1f8ca212d6045ff5b9	ConfickerAA	1a936c78281d903c0e3ec053393cd8c8	ConfickerAA	0.001	VERDADEIRO
0d0707955339f12fe598ffe0c45e5ed8	ConfickerAA	78a28dc84607a97f2f7f2150ac93b765	ConfickerAA	0.001	VERDADEIRO
1a936c78281d903c0e3ec053393cd8c8	ConfickerAA	78a28dc84607a97f2f7f2150ac93b765	ConfickerAA	0.001	VERDADEIRO
26ba4271c76b663ffa47fd3e998b0616	ConfickerAA	b1afa284cfd68c91a17ab36e5407cd4c	ConfickerAA	0.001	VERDADEIRO
35a52d24f0832d8ac6d95f2d44b5045b	ConfickerAA	9ddc0508e2873db8eb5958b4c04ee7e3	ConfickerAA	0.001	VERDADEIRO
55fa8a98c8d329978bec89437f786579	VirutAV	cbf77665b10c271a0f8d3665b240d198	VirutAV	0.001	VERDADEIRO
5dbed06e1cf05de85a9fca62d9f09b5d	ConfickerAA	b4815877466a02dfd2f0b8861d4d9c61	ConfickerAA	0.001	VERDADEIRO
6c73d39433323b436948a361f29c8340	PepexE	f024ff4176f0036f97ebc95decfd1d5e	PepexF	0.001	FALSO
6e5a212717f6ca7fb1bd064dfaa3c3d7	ConfickerX	e553228978e4157fac13877f6510a3f9	ConfickerX	0.001	VERDADEIRO
775b925e37727df6a8c2bde59f6e4c7b	ConfickerBL	c6eac8ffba8683d076dc7fabd8da1e9f	ConfickerBL	0.001	VERDADEIRO
78a28dc84607a97f2f7f2150ac93b765	ConfickerAA	1a936c78281d903c0e3ec053393cd8c8	ConfickerAA	0.001	VERDADEIRO
7a9b6845229f284e901436a949b544e8	ConfickerAA	b1afa284cfd68c91a17ab36e5407cd4c	ConfickerAA	0.001	VERDADEIRO
9613b71875be92b9725715706aa257f8	VirutAV	cbf77665b10c271a0f8d3665b240d198	VirutAV	0.001	VERDADEIRO
9ddc0508e2873db8eb5958b4c04ee7e3	ConfickerAA	35a52d24f0832d8ac6d95f2d44b5045b	ConfickerAA	0.001	VERDADEIRO
a56e77325686f944f9d0bb4c944ba1d9	ConfickerAA	b1afa284cfd68c91a17ab36e5407cd4c	ConfickerAA	0.001	VERDADEIRO
a5d5bfe1444cef3beec7140660305cda	VirutAV	55fa8a98c8d329978bec89437f786579	VirutAV	0.001	VERDADEIRO
b104c7a3b68445e36d31da658f959544	PepexF	be306aee79eb26cd5581b83e67c6bade	PepexF	0.001	VERDADEIRO
b1afa284cfd68c91a17ab36e5407cd4c	ConfickerAA	a56e77325686f944f9d0bb4c944ba1d9	ConfickerAA	0.001	VERDADEIRO
b4815877466a02dfd2f0b8861d4d9c61	ConfickerAA	5dbed06e1cf05de85a9fca62d9f09b5d	ConfickerAA	0.001	VERDADEIRO
be306aee79eb26cd5581b83e67c6bade	PepexF	b104c7a3b68445e36d31da658f959544	PepexF	0.001	VERDADEIRO
c6eac8ffba8683d076dc7fabd8da1e9f	ConfickerBL	775b925e37727df6a8c2bde59f6e4c7b	ConfickerBL	0.001	VERDADEIRO
cbf77665b10c271a0f8d3665b240d198	VirutAV	55fa8a98c8d329978bec89437f786579	VirutAV	0.001	VERDADEIRO
ce73913719776f47211e97f559a88d4f	ConfickerBL	e6dce5864be64332b7197d282bb953f2	ConfickerBL	0.001	VERDADEIRO
d827af7f090a488019622c87fcaa3dd3	DebormNAB	e42ae0e10b29f1b36e75fde65c1f788a	DebormNAB	0.001	VERDADEIRO
e42ae0e10b29f1b36e75fde65c1f788a	DebormNAB	d827af7f090a488019622c87fcaa3dd3	DebormNAB	0.001	VERDADEIRO
e553228978e4157fac13877f6510a3f9	ConfickerX	6e5a212717f6ca7fb1bd064dfaa3c3d7	ConfickerX	0.001	VERDADEIRO
e6dce5864be64332b7197d282bb953f2	ConfickerBL	ce73913719776f47211e97f559a88d4f	ConfickerBL	0.001	VERDADEIRO
e9ce3902d593b8079eb7a48474cea553	ConfickerAA	78a28dc84607a97f2f7f2150ac93b765	ConfickerAA	0.001	VERDADEIRO
f024ff4176f0036f97ebc95decfd1d5e	PepexF	6c73d39433323b436948a361f29c8340	PepexE	0.001	FALSO
f6c28ec333fe55198043f1034184ec02	ConfickerAA	9ddc0508e2873db8eb5958b4c04ee7e3	ConfickerAA	0.001	VERDADEIRO
a5e2eba8b35335df65a9ea2f56c56fbe	ConfickerGen	be2ba17405ada2f43969d114f4ac445c	ConfickerGen	0.002	VERDADEIRO
be2ba17405ada2f43969d114f4ac445c	ConfickerGen	a5e2eba8b35335df65a9ea2f56c56fbe	ConfickerGen	0.002	VERDADEIRO
bfbe3a07de66a9ca0c79c737a301c280	ConfickerAE	d089473ec3d523fd9d9414aefc73b4ce	ConfickerAE	0.002	VERDADEIRO
d089473ec3d523fd9d9414aefc73b4ce	ConfickerAE	bfbe3a07de66a9ca0c79c737a301c280	ConfickerAE	0.002	VERDADEIRO
08caa30cfea200fd422f76d06a7672cc	ConfickerAA	1a936c78281d903c0e3ec053393cd8c8	ConfickerAA	0.003	VERDADEIRO
12c0858b926d37e52b9794c4a7781faa	ConfickerAA	b9405183b603a9e4e01c54ff3cccfc32	ConfickerAA	0.003	VERDADEIRO
170bdc8e8632d001c09541863b2372a0	ConfickerAE	a613259543113ba3c86f4e6ff3ee9a90	ConfickerAE	0.003	VERDADEIRO
3c3011089708c7a49346f648f1e79384	PepexE	b0bff2664b0610ae079e52b39efdb86e	PepexE	0.003	VERDADEIRO
a613259543113ba3c86f4e6ff3ee9a90	ConfickerAE	170bdc8e8632d001c09541863b2372a0	ConfickerAE	0.003	VERDADEIRO
b0bff2664b0610ae079e52b39efdb86e	PepexE	3c3011089708c7a49346f648f1e79384	PepexE	0.003	VERDADEIRO
b9405183b603a9e4e01c54ff3cccfc32	ConfickerAA	12c0858b926d37e52b9794c4a7781faa	ConfickerAA	0.003	VERDADEIRO
13181307b2ad5064d39e2eea16711830	ConfickerAE	15761bbf7b4a38ce74a48665a9e1683f	ConfickerAE	0.004	VERDADEIRO
15761bbf7b4a38ce74a48665a9e1683f	ConfickerAE	13181307b2ad5064d39e2eea16711830	ConfickerAE	0.004	VERDADEIRO
470fd79a2f842899e2984de6bf694200	ConfickerX	d47135e14968b874889cf7ea336d75dd	ConfickerX	0.004	VERDADEIRO
d47135e14968b874889cf7ea336d75dd	ConfickerX	470fd79a2f842899e2984de6bf694200	ConfickerX	0.004	VERDADEIRO
1e01d22d81e046fad32d5695213c9fb3	ConfickerAL	59cb87dd58934f1c04cea0e351a8fb21	ConfickerAL	0.005	VERDADEIRO
501ba481d2c4b4faa1fd33b9c9265015	ConfickerAA	df7cec1843149cc9b475c314ff45dd43	ConfickerAA	0.005	VERDADEIRO
59cb87dd58934f1c04cea0e351a8fb21	ConfickerAL	1e01d22d81e046fad32d5695213c9fb3	ConfickerAL	0.005	VERDADEIRO
80f51afb3ff9a0c8c0da30258bc461e5	ConfickerAA	ded0c627a67159dfa6b38f5c121c9d8b	ConfickerAA	0.005	VERDADEIRO
ded0c627a67159dfa6b38f5c121c9d8b	ConfickerAA	80f51afb3ff9a0c8c0da30258bc461e5	ConfickerAA	0.005	VERDADEIRO
df7cec1843149cc9b475c314ff45dd43	ConfickerAA	501ba481d2c4b4faa1fd33b9c9265015	ConfickerAA	0.005	VERDADEIRO
0af49bbed7ec17b2e8b5ae7b87920715	ConfickerAK	a184507194df280881f8357c53ec2286	ConfickerAK	0.006	VERDADEIRO
22197f1ca4cbaa4b5ba57a29acf7a4b4	ConfickerAA	a2cd71ed98a1de65c15cd8adf5fa438e	ConfickerAA	0.006	VERDADEIRO
331db636e0a9cb3f59ed90fd6b1caffb	ConfickerAK	e7b67b612d0fa85ba1ccc9c063917ddf	ConfickerAK	0.006	VERDADEIRO
a184507194df280881f8357c53ec2286	ConfickerAK	0af49bbed7ec17b2e8b5ae7b87920715	ConfickerAK	0.006	VERDADEIRO
a2cd71ed98a1de65c15cd8adf5fa438e	ConfickerAA	22197f1ca4cbaa4b5ba57a29acf7a4b4	ConfickerAA	0.006	VERDADEIRO
e7b67b612d0fa85ba1ccc9c063917ddf	ConfickerAK	331db636e0a9cb3f59ed90fd6b1caffb	ConfickerAK	0.006	VERDADEIRO
5a45ba446d26faaf24e09a482aed9c35	ConfickerAA	8217439be0e8801a4a82764d1ed7df81	ConfickerAA	0.007	VERDADEIRO
8217439be0e8801a4a82764d1ed7df81	ConfickerAA	5a45ba446d26faaf24e09a482aed9c35	ConfickerAA	0.007	VERDADEIRO
cc69432ca56c00400eb0acc40c5aa361	ConfickerBL	775b925e37727df6a8c2bde59f6e4c7b	ConfickerBL	0.007	VERDADEIRO
1a07f64eee7fe8309dc96ddfb7e3a1d9	ConfickerAA	2bb807e91907956c4da88fdfddd9c855	ConfickerAA	0.008	VERDADEIRO
2bb807e91907956c4da88fdfddd9c855	ConfickerAA	1a07f64eee7fe8309dc96ddfb7e3a1d9	ConfickerAA	0.008	VERDADEIRO
7cbf9b7ac8a65ed4a578821663905f38	ConfickerAL	a265f9fafacb24e2d7859eef6b9e645f	ConfickerAL	0.008	VERDADEIRO
7cf59b6e0605d3c354dcc8c693d65767	ConfickerX	f021ad6f27029c1ae0913ccc132dff15	ConfickerGen	0.008	FALSO
a265f9fafacb24e2d7859eef6b9e645f	ConfickerAL	7cbf9b7ac8a65ed4a578821663905f38	ConfickerAL	0.008	VERDADEIRO
d1e8f4fa991b337e37221ff3c72838b3	ConfickerAA	f9215c265c68c27b4359925c0268b0cd	ConfickerAA	0.008	VERDADEIRO
f021ad6f27029c1ae0913ccc132dff15	ConfickerGen	7cf59b6e0605d3c354dcc8c693d65767	ConfickerX	0.008	FALSO
f9215c265c68c27b4359925c0268b0cd	ConfickerAA	d1e8f4fa991b337e37221ff3c72838b3	ConfickerAA	0.008	VERDADEIRO
12c0496600a94ab4ddba5d9ef1fc74cf	ConfickerX	b53b1f1a00c8eb0551a2d3d68fa3e6d6	ConfickerX	0.009	VERDADEIRO
b53b1f1a00c8eb0551a2d3d68fa3e6d6	ConfickerX	12c0496600a94ab4ddba5d9ef1fc74cf	ConfickerX	0.009	VERDADEIRO

APÊNDICE B

Listagens dos códigos empregados neste trabalho.

Listagem 5.1: Código I: comparação de *hashes* perceptivas

```
# -*- coding: utf-8 -*-
"""
@author: JP
"""
#imports
import imagehash as ihash
from PIL import Image
import csv
import scipy
from scipy import spatial
import time
import numpy as np
import os

#inicializacao das variaveis
start_time = time.time()
class_dict = {}

with open(r"D:\TCC\codigos\1_Working\bd.csv", 'r') as csvfile:
        reader = csv.DictReader(csvfile, delimiter = ';')
        for linha in reader:
                class_dict[linha['md5']] = {'familia' : linha['familia'],
                    'variante' : linha['variante'], 'vt' : linha['tipo']}
csvfile.close()


os.chdir(r"D:\TCC\resources\atual")

def hamming_distance(image, image2):
    #dois arrays representando imagens
    score =scipy.spatial.distance.hamming(image, image2)
    return score #porcentagenm de diferenca

i=0
n = 100

```

```
36 for md5 in class_dict:
37     hash_ = ihash.whash(Image.open(md5))
38     class_dict[md5]['whash'] =  np.hstack(hash_.hash)
39     hash_ = ihash.average_hash(Image.open(md5))
40     class_dict[md5]['ahash'] =  np.hstack(hash_.hash)
41     hash_ = ihash.dhash(Image.open(md5))
42     class_dict[md5]['dhash'] =  np.hstack(hash_.hash)
43     hash_ = ihash.phash(Image.open(md5))
44     class_dict[md5]['phash'] =  np.hstack(hash_.hash)
45
46     print i
47     i +=1
48 tempo_classificacao_ok = time.time() - start_time
49 print "classificacao levou " + time.strftime("%H:%M:%S", time.gmtime(
       tempo_classificacao_ok))
50
51 variante_a = 0
52 variante_d= 0
53 variante_p= 0
54 variante_w= 0
55
56 familia_a= 0
57 familia_d= 0
58 familia_p= 0
59 familia_w= 0
60
61 tipo_a = 0
62 tipo_d = 0
63 tipo_p = 0
64 tipo_w = 0
65
66 erro_a = 0
67 erro_d = 0
68 erro_p = 0
69 erro_w = 0
70
71 tempo_comparacao = time.time()
72
73 for ham in class_dict:
74     maior_a = 10
75     maior_d = 10
76     maior_p = 10
77     maior_w = 10
78
79     for ham2 in class_dict:
80         hamming_a = hamming_distance(class_dict[ham]['ahash'],class_dict[
               ham2]['ahash'])
81         hamming_d = hamming_distance(class_dict[ham]['dhash'],class_dict[
               ham2]['dhash'])
```

```
82          hamming_p = hamming_distance(class_dict[ham]['phash'],class_dict[
                ham2]['phash'])
83          hamming_w = hamming_distance(class_dict[ham]['whash'],class_dict[
                ham2]['whash'])
84
85          if (hamming_a < maior_a and hamming_a != 0):
86              maior_a = hamming_a
87              resultado_a = ham2
88
89          if (hamming_d < maior_d and hamming_d != 0):
90              maior_d = hamming_d
91              resultado_d = ham2
92
93          if (hamming_p < maior_p and hamming_p != 0):
94              maior_p = hamming_p
95              resultado_p = ham2
96
97          if (hamming_w < maior_w and hamming_w != 0):
98              maior_w = hamming_w
99              resultado_w = ham2
100
101     if (class_dict[ham]['variante'] == class_dict[resultado_a]['variante'
            ]):
102         variante_a +=1
103         familia_a +=1
104         tipo_a +=1
105     elif(class_dict[ham]['familia'] == class_dict[resultado_a]['familia'
            ]):
106         familia_a +=1
107         tipo_a +=1
108     else:
109         erro_a +=1
110
111     if (class_dict[ham]['variante'] == class_dict[resultado_d]['variante'
            ]):
112         variante_d +=1
113         familia_d +=1
114         tipo_d +=1
115     elif(class_dict[ham]['familia'] == class_dict[resultado_d]['familia'
            ]):
116         familia_d +=1
117         tipo_d +=1
118     else:
119         erro_d +=1
120
121     if (class_dict[ham]['variante'] == class_dict[resultado_p]['variante'
            ]):
122         variante_p +=1
123         familia_p +=1
```

```
124             tipo_p +=1
125         elif(class_dict[ham]['familia'] == class_dict[resultado_p]['familia'
                ]):
126             familia_p +=1
127             tipo_p +=1
128         else:
129             erro_p +=1
130 
131         if (class_dict[ham]['variante'] == class_dict[resultado_w]['variante'
                ]):
132             variante_w +=1
133             familia_w +=1
134             tipo_w +=1
135         elif(class_dict[ham]['familia'] == class_dict[resultado_w]['familia'
                ]):
136             familia_w +=1
137             tipo_w +=1
138         elif(class_dict[ham]['vt'] == class_dict[resultado_w]['vt']):
139             tipo_w +=1
140         else:
141             erro_w +=1
142 
143         file = open(r'D:\TCC\resultados\db_8.txt','a')
144         file.write("aHash %s %s %s %s %s %s %f dHash %s %s %s %s %s %s %f
               pHash %s %s %s %s %s%s %f wHash %s %s %s %s %s %s %f" % (ham,
               class_dict[ham]['familia'],class_dict[ham]['variante'],resultado_a
               ,class_dict[resultado_a]['familia'],class_dict[resultado_a]['
               variante'],maior_a, ham,class_dict[ham]['familia'],class_dict[ham
               ]['variante'],resultado_d,class_dict[resultado_d]['familia'],
               class_dict[resultado_d]['variante'],maior_d, ham,class_dict[ham]['
               familia'],class_dict[ham]['variante'],resultado_p,class_dict[
               resultado_p]['familia'],class_dict[resultado_p]['variante'],
               maior_p, ham,class_dict[ham]['familia'],class_dict[ham]['variante'
               ],resultado_w,class_dict[resultado_w]['familia'],class_dict[
               resultado_w]['variante'],maior_w))
145         file.write('\n')
146         file.close()
147 
148 tempo_comparacao_ok = time.time() - tempo_comparacao
149 print "comparacao levou " + time.strftime("%H:%M:%S", time.gmtime(
        tempo_comparacao_ok))
150 
151 elapsed_time = tempo_comparacao_ok + tempo_classificacao_ok
152 print "Quantidade de amostras %d " % len(class_dict)
153 print "Todo o programa levou " + time.strftime("%H:%M:%S", time.gmtime(
        elapsed_time))
```

Listagem 5.2: Código II: comparação de imagens

```
1 
```

```
2  import numpy as np
3  from skimage.measure import compare_ssim as ssim
4  import cv2
5  import os
6  import csv
7  import time
8
9  start_time = time.time()
10 class_dict = {}
11
12 with open(r"D:\TCC\codigos\1_Working\atual.csv", 'r') as csvfile:
13         reader = csv.DictReader(csvfile, delimiter = ';')
14         for linha in reader:
15                 class_dict[linha['md5']] = {'familia' : linha['familia'],
                       'variante' : linha['variante'], 'vt' : linha['tipo']}
16 csvfile.close()
17
18
19 def mmse(imageA, imageB):
20     # the 'Mean Squared Error' between the two images is the
21     # sum of the squared difference between the two images;
22     # NOTE: the two images must have the same dimension
23     err = np.sum((imageA.astype("float") - imageB.astype("float")) ** 2)
24     err /= float(imageA.shape[0] * imageA.shape[1])
25     return err
26
27 def compare_images(imageA, imageB):
28     # compute the mean squared error and structural similarity
29     # index for the images
30     m = mmse(imageA, imageB)
31     s = ssim(imageA, imageB,multichannel=True)
32     return m,s
33
34 variante_mse = 0
35 familia_mse = 0
36 variante_ssim = 0
37 familia_ssim = 0
38 erro_mse = 0
39 erro_ssim = 0
40 i = 0
41 tempo_comparacao = time.time()
42 os.chdir(r"D:\TCC\resources\atual")
43
44 for amostra in class_dict:
45
46     mse = 9999
47     ssim_ = -1
48     imagem = cv2.imread(amostra)
49     imagem = cv2.cvtColor(imagem, cv2.COLOR_BGR2GRAY)
```

```
50
51      for busca in class_dict:
52
53          imagem2 = cv2.imread(busca)
54          imagem2 = cv2.cvtColor(imagem2, cv2.COLOR_BGR2GRAY)
55          m,s = compare_images(imagem,imagem2)
56
57          if m < mse:
58              mse = m
59              classificacao_mse = busca
60          if s > ssim_:
61              ssim_ = s
62              classificaco_ssim = busca
63
64      if (class_dict[amostra]['variante'] == class_dict[classificacao_mse][
            'variante']):
65          variante_mse +=1
66      elif(class_dict[amostra]['familia'] == class_dict[classificacao_mse][
            'familia']):
67          familia_mse +=1
68      else:
69          erro_mse +=1
70
71      if (class_dict[amostra]['variante'] == class_dict[classificaco_ssim][
            'variante']):
72          variante_ssim +=1
73      elif(class_dict[amostra]['familia'] == class_dict[classificaco_ssim][
            'familia']):
74          familia_ssim +=1
75      else:
76          erro_ssim +=1
77      file = open(r'D:\TCC\codigos\1_Working\MSE SSIM\DB.txt','a')
78      file.write("ssim %s %s%s %s %s%s %.6f mmse %s %s%s %s %s%s %.6f" % (
            amostra,class_dict[amostra]['familia'],class_dict[amostra]['
            variante'],classificaco_ssim,class_dict[classificaco_ssim]['
            familia'],class_dict[classificaco_ssim]['variante'],ssim_, amostra
            ,class_dict[amostra]['familia'],class_dict[amostra]['variante'],
            classificacao_mse,class_dict[classificacao_mse]['familia'],
            class_dict[classificacao_mse]['variante'],mse))
79      file.write('\n')
80      file.close()
81      print i
82      i +=1
83
84  tempo_comparacao_ok = time.time() - tempo_comparacao
85  print "comparacao levou " + time.strftime("%H:%M:%S", time.gmtime(
        tempo_comparacao_ok))
86
87
```

```
88 elapsed_time = time.time() - start_time
89 print "Quantidade de amostras %d " % len(class_dict)
90 print "Todo o programa levou " + time.strftime("%H:%M:%S", time.gmtime(
       elapsed_time))
```