Instituto de Computação
Universidade Estadual de Campinas

BehEMOT - Um sistema híbrido de análise de *malware*

Dario S. Fernandes Filho

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Dario S. Fernandes Filho e aprovada pela Banca Examinadora.

Campinas, 06 de outubro de 2011.

Paulo Lício de Geus (Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA POR
MARIA FABIANA BEZERRA MÜLLER - CRB8/6162
BIBLIOTECA DO INSTITUTO DE MATEMÁTICA, ESTATÍSTICA E
COMPUTAÇÃO CIENTÍFICA - UNICAMP

F391b

Fernandes Filho, Dario Simões, 1986-
BehEMOT : um sistema híbrido de análise de malware / Dario Simões Fernandes Filho. - Campinas, SP : [s.n.], 2011.

Orientador: Paulo Lício de Geus.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Crime por computador. 2. Sistemas de segurança. 3. Comportamento - Avaliação. I. Geus, Paulo Lício de, 1956-. II. Universidade Estadual de Campinas. Instituto de Computação. III.Título.

Informações para Biblioteca Digital

Título em inglês: BehEMOT: a hybrid malware analysis system
Palavras-chave em inglês:
Computer crimes
Computer security
Behavioral analysis
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Paulo Lício de Geus [Orientador]
Edmundo Roberto Mauro Madeira
Adriano Mauro Cansian
Data da defesa: 06-10-2011
Programa de Pós-Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 06 de outubro de 2011, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Adriano Mauro Cansian
DCCE / UNESP

Prof. Dr. Edmundo Roberto Mauro Madeira
IC / UNICAMP

Prof. Dr. Paulo Lício de Geus
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

BehEMOT - Um sistema híbrido de análise de *malware*

Dario S. Fernandes Filho

Outubro de 2011

Banca Examinadora:

- Paulo Lício de Geus (Orientador)
- Edmundo Roberto Mauro Madeira
- Adriano Mauro Cansian
- Rogério Drummond Burnier Pessoa de Mello Filho (Suplente)
- Carlos Alberto Maziero (Suplente)

Resumo

O aumento no número de operações financeiras ocorrendo na Internet impulsionou o crescimento nos ataques a usuários conectados. Estes ataques normalmente são feitos com o uso de malware, software que realiza ações maliciosas na máquina do usuário, tais como interceptação de dados sensíveis, por exemplo, senhas e números de cartões de crédito. A fim de minimizar o comprometimento por malware, são utilizados mecanismos antivírus, software usados para encontrar e remover *malware*. Tal detecção normalmente é feita através de assinaturas – strings que auxiliam na identificação – ou heurísticas. Entretanto, essa abordagem pode ser facilmente subvertida, tornando a identificação dos *malware* ineficaz. Para evitar este problema, é usado um outro tipo de abordagem de detecção, onde o comportamento do binário no sistema é analisado. O trabalho proposto visa desenvolver um protótipo de um sistema de análise de *malware* que poderá gerar perfis comportamentais, os quais podem servir de insumo para ferramentas de detecção de *malware*.

Abstract

The rise in the number of financial operations through the internet boosted the increase in the attacks to connected users. These attacks are normally made by malware, software that make malicious actions in the user machine, such as interception of sensitive data, like passwords and card numbers. To minimize the compromise by malware, anti-virus mechanisms are frequently used, software that usually finds and removes malware. Such detection are normally made through signatures – strings that help in the identification – or heuristics. However, this approach can be easily subverted, making the identification of malware ineffective. To avoid this problem, it's used another detection approach, where the binary behavior is analyzed. The proposed work aims to develop a prototype of a malware analysis system which may generate behavior profiles, which can serve as an input to malware detection tools.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

O uso da Internet para a realização de transações financeiras como *online banking* e compra de produtos tem aumentado consideravelmente nos últimos anos. Para realizar essas ações é necessário o uso de dados sensíveis, tais como senhas diversas e números de cartões de crédito, os quais são transmitidos geralmente de forma segura através da utilização de algum tipo de protocolo criptográfico (ex.: SSL).

Tendo em vista que na maioria das vezes não é trivial atacar a criptografia utilizada no tráfego de rede para obter os dados mencionados, os atacantes começaram a centralizar os esforços nos sistemas das vítimas, de onde as informações podem ser acessadas ou capturadas antes da atuação de qualquer mecanismo de confidencialidade. Uma forma muito comum e eficiente de comprometer as máquinas dos usuários é com o uso de *malware* [1], um tipo particular de *software* que realiza operações danosas no sistema sem o conhecimento do usuário. Os objetivos e motivações dos ataques por *malware* variam desde o roubo de dados pessoais e/ou sigilosos para fins lucrativos ou de espionagem industrial, até a formação de redes de sistemas comprometidos para perpetrar ataques contra outros sistemas, armazenar conteúdo impróprio ou ilícito e servir como um anonimizador das atividades do atacante.

Para que um *malware* seja instalado no sistema do usuário, o atacante pode lançar mão de diversas abordagens, em geral envolvendo iludir a vítima a se infectar (engenharia social). É comum o atacante instigar a curiosidade das vítimas com mensagens de *e-mail* com notícias chamativas, ou mesmo através de mensagens não solicitadas (*spam*) com propagandas de medicamentos ou produtos falsos, levando-as a acessar endereços na Internet com conteúdo malicioso, culminando na instalação do *malware* no sistema do usuário. Após a execução, o programa malicioso instalado provê inúmeras possibilidades aos atacantes, como por exemplo o controle remoto do sistema, a execução de comandos arbitrários, o roubo de informações e arquivos, a alteração ou destruição de dados diversos e, ainda, o lançamento de ataques de forma anônima (em relação à identidade do real

atacante).

De acordo com pesquisas recentes, a quantidade de exemplares de *malware* observados no período entre janeiro a setembro de 2010 foi de aproximadamente 14 milhões, um valor superior em um milhão ao mesmo período do ano passado. [2]. Isso torna indispensável o uso de medidas de combate aos problemas com programas maliciosos. Uma solução muito comum é o uso de mecanismos antivírus, programas que monitoram um sistema constantemente em busca de arquivos infectados ou com comportamento suspeito. Existe uma grande diversidade de programas antivírus disponíveis atualmente, os quais realizam a detecção de *malware* através do uso de assinaturas—cadeias sequenciais de bytes que identificam um exemplar de *malware*. As assinaturas dos antivírus são geradas a partir de porções representativas do programa malicioso e podem, por exemplo, ser um conjunto de instruções ou uma sequência de caracteres contidas neste. Para gerá-las, é necessária a análise prévia do *malware* por um especialista, a fim de que seja definido qual será o padrão de identificação utilizado. As assinaturas são armazenadas em uma base de assinaturas, isto é, um arquivo no qual o antivírus busca as informações sobre os *malware* conhecidos.

Para dificultar a detecção de *malware* através de assinaturas, os atacantes fazem uso de *packers* [3], que são rotinas que visam comprimir ou cifrar o código binário de um programa de modo a ofuscá-lo e tornar a assinatura que o detectava ineficiente. A aplicação de *packers* e a utilização de outros métodos para ofuscar o código real do *malware* faz com que surjam muitas variantes, consequentemente aumentando a quantidade de assinaturas necessárias para identificar uma mesma família de *malware*. Tal efeito pode ser percebido em informações relatadas por fabricantes de antivírus em 2009, ano no qual a quantidade de assinaturas aumentou 71% em relação a 2008 [4]. Algumas soluções foram propostas para remover automaticamente os *packers* de programas maliciosos [5, 6] e facilitar a identificação destes por assinaturas de mecanismos antivírus. Entretanto, a sobrecarga gerada para a remoção pode ser muito grande e o método de remoção pode não ser aplicável a todos os tipos de *packer* existentes, tornando o processo de *unpacking* ineficaz.

Um outro problema grave dos antivírus baseados em assinaturas é a falta de uma proteção efetiva ao usuário na janela de tempo entre o surgimento da variante do *malware* e a disponibilização da assinatura que o detecte. Com o aumento no número de *malware* em circulação [4, 2], a necessidade de novas assinaturas em um espaço de tempo cada vez menor torna-se um problema evidente. Essa demanda obriga a atualização frequente no banco de assinaturas, impactando em seu tamanho e exigindo uma eficiência maior no processo de identificação, que tem que lidar com consultas a um banco cada vez maior. Devido a geração de novas assinaturas ser um trabalho custoso, dado que requer a intervenção de um especialista, a automatização torna-se uma solução interessante. Alguns trabalhos têm sido desenvolvidos para gerar assinaturas de forma automatizada, porém

eles apresentam alguns problemas, seja para lidar com *malware* nos quais foram aplicados *packers* [7], ou por não levar em conta o código completo da execução do programa, limitando a análise ao cabeçalho do binário malicioso [8]. Apesar de todos os problemas citados anteriormente, a identificação de *malware* baseada em assinaturas de antívirus ainda é amplamente realizada.

Uma alternativa que tem sido muito utilizada na identificação de *malware* é a análise do seu comportamento no sistema da vítima. Esse tipo de identificação é realizada observando-se o conjunto de chamadas de sistema (*syscalls*) efetuadas pelo programa malicioso durante sua execução. Essas chamadas são responsáveis por ações que requerem um nível de acesso privilegiado ao *kernel* do sistema operacional, servindo de interface entre este e o *malware*. Dado que uma grande parte das atividades realizadas por um programa se utiliza de *syscalls*, torna-se possível observar de forma minuciosa as ações que foram efetuadas para completar sua execução. Entretanto, é necessário identificar quais dessas ações compõem o comportamento de grupos de *malware* para, dessa forma, verificar se um programa qualquer ainda não identificado apresenta comportamento semelhante. Como é comum que vários exemplares de *malware* sejam variações de um programa malicioso original, seus comportamentos tendem a ser parecidos mesmo com o uso de *packers* e, portanto, espera-se que o armazenamento de poucos comportamentos leve a identificação de muitos exemplares distintos.

Assim, o trabalho proposto para esta dissertação visa o desenvolvimento do protótipo para análise dinâmica de *malware* chamado *BehEMOT*[1], o qual irá monitorar as atividades realizadas por um programa (no nível das chamadas de sistema) durante sua execução em um sistema operacional *Windows*. As atividades monitoradas servem para a extração do comportamento do programa analisado e, agregadas ao tráfego de rede capturado durante a execução, tornam possível a geração de perfis que caracterizem *malware* e possam ajudar em futuras identificações. As contribuições dadas por este trabalho são:

- A implementação de BehEMOT como um *driver* de *kernel* torna a ferramenta flexível, permitindo sua utilização em outras soluções de segurança (por exemplo, monitoração de modificações no estado do sistema alvo por ataques via Web) e em vários tipos de ambiente, tais como máquinas virtuais, máquinas reais e emuladores. Além disso, por não utilizar técnicas intrusivas na monitoração do comportamento dos programas, BehEMOT evita que o exemplar de *malware* perceba que está sendo monitorado e altere seu fluxo execução.

- A arquitetura proposta para o uso de BehEMOT foi projetada com o intuito de se utilizar um ambiente misto (análise emulada seguida de análise em máquina real, caso ocorram problemas), tornando ineficientes as técnicas de evasão e anti-análise

[1] *Behavior Evaluation from Malware Observation Tool*

utilizadas por classes específicas de *malware* para detecção de máquinas virtuais e emuladores, bem como possibilitando à ferramenta a análise de uma variedade maior de *malware* quando comparada às abordagens de análise dinâmica existentes.

- Dado que as atividades envolvendo rede são capturadas de duas formas—dentro do ambiente de análise, através da monitoração das *syscalls* de envio e recebimento de dados na interface de rede; fora do ambiente de análise, através da captura de tráfego em um *gateway* externo, é possível obter informações acerca do comportamento de *rootkits*[2] que porventura comuniquem-se através da interface de rede.

O restante desta dissertação foi dividido em três partes: a revisão bibliográfica, que discute as abordagens utilizadas em sistemas de análise conhecidos, a descrição do sistema, que contém os detalhes relativos à construção da ferramenta e a arquitetura do sistema proposto e, finalmente, os capítulos referentes aos testes realizados e resultados obtidos, conclusão da dissertação e trabalhos futuros que podem ser derivados deste.

[2] *Rootkit* é um tipo específico de *malware* cuja característica principal é a ocultação de suas atividades no sistema da vítima com a finalidade de não ser detectado por mecanismos de segurança.

Capítulo 2

Revisão Bibliográfica

O processo de análise de código malicioso pode levar a um maior entendimento sobre as ações realizadas por *malware* nos sistemas infectados. Pode-se dividir a análise de *malware* em duas categorias, estática e dinâmica, cada qual possuindo características próprias que podem indicar (e justificar) quando é necessário utilizar uma ou outra.

Na **análise estática**, o objeto de análise é o código binário do programa malicioso, o qual é composto por suas instruções *assembly* e contém a sequência de ações que devem ser realizadas na execução. Com esse tipo de análise é possível extrair informações que estejam em claro no código, ou seja, sem ofuscação ou criptografia. Em geral, tais informações tratam-se de cadeias de caracteres como, por exemplo, endereços de *sites* Web (*URL*), de *e-mail* ou de rede (IP), os quais podem levar a algum recurso dominado pelo autor do *malware*. Como apenas o código do programa é observado, não é necessário executá-lo para obter as informações mencionadas. Isso torna a análise estática vantajosa do ponto de vista da integridade do sistema de análise, pois as atividades de ataque do *malware* em observação são inócuas, uma vez que não há execução. Entretanto, caso o programa sob análise seja ofuscado ou cifrado (por exemplo, através de um *packer*), torna-se difícil a extração das informações do código do *malware*. Com isso, a fim de impossibilitar a aplicação de técnicas de análise estática, os desenvolvedores de *malware* têm cada vez mais se utilizado de ferramentas de *packing*.

Packers são mecanismos utilizados em arquivos executáveis cuja finalidade é compactá-los, diminuindo seu tamanho original e, consequentemente, ocupando um menor espaço em disco. Além desta aplicação, *packers* também são utilizados para ofuscar códigos de programas, dado que após a ofuscação é gerado um novo código binário com instruções *assembly* diferentes do original [3]. Inicialmente, o intuito desta funcionalidade era a utilização por programas benignos para proteção do código, a fim de evitar que técnicas de engenharia reversa fossem aplicadas à violação de direitos autorais, modificação não-autorizada ou pirataria (*cracking*). Entretanto, desenvolvedores de *malware*

começaram a utilizar *packers* para dificultar a análise estática e inviabilizar a detecção de seus programas através de antivírus por assinaturas, haja visto que a ofuscação impede a correspondência de partes do código malicioso com as assinaturas de detecção, possibilitando a atuação do *malware* de maneira indetectável. Para contornar esta situação, há diversas técnicas de *unpacking* [9], [10], [6] que extraem o código original através da remoção do *packer*, porém, tal processo pode ser demorado e impactar diretamente no tempo necessário para se realizar a análise. Paralelamente a isso, não há uma abordagem universal e genérica, o que leva à necessidade de identificação do *packer* ou remoção manual do mesmo.

Além disso, mesmo que o binário não esteja ofuscado com um *packer*, a análise estática apresenta outra limitação, esta relacionada às respostas produzidas pelo sistema atacado durante uma determinada execução do *malware*. Assim, analisando-se somente o código, não é possível verificar como o programa malicioso interage com o sistema (e como esse responde às ações maliciosas), pois tal interação não existe. Por fim, como podem existir inúmeros caminhos (ou fluxos) de execução diferentes em um mesmo programa, a análise estática pode ser inviável, devido à complexidade requerida para se seguir todos esses fluxos [11].

Já na **análise dinâmica** o processo é realizado através da execução do *malware*, o que torna possível contornar alguns dos problemas encontrados na análise estática. Nesse caso, o código ofuscado não é mais um empecilho devido à execução implicar na remoção dinâmica do *packer*, fazendo com que as instruções efetuadas no sistema de análise correspondam àquelas do programa original, sem o *packer*. O ponto principal da análise dinâmica é o efeito produzido pela execução do *malware*, que causa o comprometimento do sistema no qual a análise foi realizada. Quando o *malware* atua no sistema de forma a realizar as ações que compõem seu comportamento malicioso, é possível monitorar e capturar as interações entre ambos (*malware* e sistema de análise). Entretanto, fazem-se necessários métodos de controle e gerenciamento do ambiente de execução, para evitar que ataques sejam lançados contra outros sistemas e redes, e para permitir que o ambiente de análise retorne a um estado íntegro que não contenha nenhum dos efeitos do comprometimento. Este último é um requisito essencial, pois caso contrário as análises futuras irão devolver dados incorretos e não confiáveis, que podem ser frutos da atividade de algum exemplar de *malware* executado anteriormente.

Assim, a captura dos dados produzidos durante a execução de um dado *malware* em um sistema de análise dinâmica pode ser feita de várias formas, as quais se diferenciam pela maneira como as ações efetuadas são interceptadas. Pode-se dividir as técnicas de interceptação utilizadas em análise dinâmica principalmente pelo nível de privilégio requerido no componente responsável pela captura, que pode ser o mesmo do *malware* sob análise (nível de usuário) ou um nível que necessita um maior privilégio (*kernel* do sis-

tema). Os níveis de privilégio são utilizados para diferenciar as permissões possíveis para cada processo e usuário, de forma que cada um deles seja limitado somente ao necessário para completar sua tarefa. Essa compartimentalização de permissões define o princípio de privilégio mínimo, o qual torna possível evitar que acessos indevidos ocorram no sistema [12]. O princípio do privilégio mínimo é utilizado nos sistemas operacionais para diferenciar o privilégio de execução dos seus componentes, sendo que os níveis possíveis são chamados de *Ring Levels* e variam de 0—o mais privilegiado—até 3, com privilégios limitados.

Sistemas operacionais *Windows XP*, presentes nos principais sistemas de análise dinâmica de *malware*, usam apenas dois níveis: o nível 3, denominado *user space* ou modo de usuário e que contém as aplicações gerais com permissões de acesso controlado e privilégio reduzido, e o 0, chamado de *kernel space* ou modo de kernel, onde são executadas as aplicações com privilégio mais elevado que têm acesso a todos os dispositivos do sistema operacional [13]. Como exemplo de uma aplicação de modo de usuário, pode-se citar um editor de texto, um navegador Web ou qualquer outra aplicação executada pelo usuário. Aplicações de nível de *kernel* são responsáveis, na maioria das vezes, por fazer a interface entre os dispositivos do sistema e os programas que rodam em nível de usuário. Programas de níveis mais privilegiados têm total acesso aos de menor privilégio, isto é, aqueles que são executados em nível de *kernel* têm total controle sobre as aplicações de nível de usuário. Além disso, quem executa no nível de usuário não possui acesso direto ao nível do *kernel*, devendo este ser feito através de chamadas de sistema, que são métodos disponibilizados pelo sistema operacional para que programas com menor privilégio tenham acesso aos níveis mais privilegiados [14].

Logo, caso a captura das ações executadas pelo *malware* ocorra no mesmo nível de privilégio de sua execução, é possível que este utilize-se de técnicas triviais para detectar a monitoração. Com isso, o comportamento apresentado pelo programa malicioso pode ser diferente de seu comportamento habitual. Isto ocorre pois, ao perceber que está sendo monitorado, o *malware* pode apresentar um comportamento alternativo composto majoritariamente por ações insuspeitas. Embora existam meios para se tentar enganar o *malware* de forma que este não perceba que está sendo monitorado, trata-se de um processo complicado, limitado [15] e que também pode ser subvertido. Desse modo, fica evidente que o componente responsável por capturar as ações executadas pelo programa sob análise deve ser implementado em um nível de privilégio superior, minimizando a oportunidade de acesso por parte do *malware* e evitando, assim, possíveis subversões do sistema e análises com resultados incorretos.

Dadas as limitações apresentadas na extração de comportamento de *malware* ao se utilizar técnicas de análise estática, definiu-se que o escopo desta dissertação abrange, dentre outras coisas, o desenvolvimento de uma ferramenta para análise dinâmica. Nas

seções a seguir serão detalhadas as principais técnicas empregadas na análise dinâmica de *malware*, bem como serão citados os sistemas e ferramentas que as utilizam, de forma a fazer um levantamento bibliográfico do estado da arte e justificar a escolha da técnica empregada em BehEMOT.

2.1 Virtual Machine Introspection

Virtual Machine Introspection (VMI) é uma técnica que consiste em criar uma camada intermediária entre o ambiente de análise (*guest*) e o ambiente de processamento (*host*). Através do uso de VMI é possível obter as ações que foram executadas pelo *malware* durante a análise dinâmica, sem que haja qualquer interferência dentro do ambiente no qual este está sendo executado. Programas para emulação e virtualização, tais como Qemu [16], VMWare [17] e VirtualBox [18] possibilitam a aplicação desta técnica, dado que estes implementam a camada intermediária de maneira nativa, fazendo assim uma distinção entre o ambiente real (ou sistema *host*) e o emulado/virtualizado (ou o *guest*). Nesta seção serão explicados os conceitos de virtualização e emulação, e como a técnica de VMI pode ser aplicada a eles.

Emulação é um termo utilizado na área de computação para descrever o modo de operação de um *software* desenvolvido para simular um determinado *hardware* [19], como por exemplo um processador específico diferente do que está executando o emulador. Na análise de *malware*, este tipo de *software* é utilizado para simular uma máquina, para que seja possível a instalação do sistema operacional que será utilizado no processo de análise. Outra utilidade do emulador é isolar o ambiente de análise, fazendo com que as ações efetuadas por um exemplar de *malware* durante sua execução não contaminem o ambiente real. Como as modificações ocasionadas pela execução do *malware* ocorrem somente no sistema operacional emulado, a máquina *host* não sofre nenhum dano. Como exemplo de emulador muito utilizado para este fim, pode-se citar o Qemu [16]—uma ferramenta de código aberto e de fácil utilização com o qual é possível emular o processador, o disco rígido e demais dispositivos do sistema de forma que seja possível instalar vários tipos de sistema operacional.

Virtualização, de modo similar à emulação, é utilizada para simular uma máquina, tornando possível a instalação e execução de vários sistemas operacionais em paralelo com o mesmo *hardware*. Porém, na virtualização as instruções são executadas no *hardware* real da máquina, ao contrário do que ocorre na emulação, na qual as instruções são executadas em um processador emulado. Isto torna a virtualização mais rápida em relação à emulação, pois as instruções do *guest* ficam sob responsabilidade do *hardware* do *host*. A limitação da virtualização é que esta possibilita somente a instalação de sistemas cuja arquitetura é a mesma do *host*. Quando se utiliza um virtualizador para análise de

malware, as funcionalidades são parecidas com as do emulador: há o isolamento entre os ambientes dos sistemas *host* e *guest*. Nesse caso, o programa responsável pela virtualização, acrescenta uma camada adicional entre o ambiente real e o de análise chamada de *Virtual Machine Monitor*. Esta camada realiza a abstração do *hardware* real para as máquinas virtuais, executando suas ações de forma que sejam percebidas somente dentro do ambiente virtual [20].

Tanto a virtualização quanto a emulação fazem uma distinção entre o ambiente onde o *malware* é executado e o real. Para simplificar a forma de mencionar tais ambientes, a partir de agora serão utilizados os termos *guest* e *host*, sendo que o primeiro identifica o sistema operacional virtualizado ou emulado utilizado na análise dinâmica e o último identifica o sistema base que executa o emulador ou virtualizador.

O programa de virtualização/emulação responde ao sistema *guest* da mesma forma que os dispositivos físicos (processador, disco rígido, placas de rede e vídeo etc) responderiam, sem entretanto comprometer o *host* no qual ele está sendo executado. Essa transparência faz com que o *host* tenha total controle sobre o *guest*, podendo inclusive observar em tempo real o estado dos diversos recursos da máquina onde o *malware* está sendo executado, como memória e CPU, por exemplo.

Desta forma, torna-se trivial a obtenção de informações a respeito da execução do *malware* de maneira externa ao *guest*, bastando que se modifique o *software* responsável por executar a emulação/virtualização para que este realize a captura dos dados. A modificação de um programa de virtualização ou emulação com o objetivo de se obter informações internas ao *guest* a partir do sistema *host* é chamada de VMI e, com a utilização desta técnica é possível alcançar um nível de privilégio adicional na camada de abstração intermediária entre o *host* e o *guest*. Uma das características mais interessantes da VMI é que esta torna possível a análise de *malware* cuja execução ocorre no nível do *kernel*, tais como os *rootkits*. As técnicas comumente utilizadas por *rootkits* para esconder ou alterar estruturas internas do sistema operacional atacados [21] podem inviabilizar sua detecção e monitoração por mecanismos de segurança ou outros métodos de análise, como por exemplo, *hooking* (Seção 2.2).

Para ilustrar a técnica de VMI, um tipo de informação que pode ser capturada do sistema *guest* são as *syscalls* que o *malware* executou durante a análise. Um método muito utilizado para identificar a ocorrência de uma *syscall* baseia-se na leitura do valor contido no registrador `SYSENTER_EIP_MSR` do processador. Este registrador é utilizado quando ocorre uma instrução do tipo "`SYSENTER`", que indica que uma chamada de sistema deve ser feita. Quando a chamada é efetuada, o sistema realiza a troca de contexto entre o espaço de usuário e o espaço de *kernel*, permitindo finalmente a execução da *syscall*.

Uma forma de identificar qual *syscall* está sendo invocada é através da leitura do valor contido no registrador `EAX`. No momento em que a instrução `SYSTENTER` for executada, o

registrador EAX armazena um valor utilizado para se encontrar o endereço da *syscall* que se quer realizar. Esse valor corresponde ao índice de uma tabela que contém os endereços de todas as *syscalls* possíveis no sistema operacional. Em sistemas *Windows*, esta tabela corresponde a uma estrutura que atende pelo nome de *System Service Dispatch Table*. Os parâmetros utilizados para compor a *syscall* podem ser obtidos através de verificações nos registradores do processador e na memória do sistema, no momento em que a chamada estiver sendo executada.

Um sistema de análise dinâmica bem conhecido e disponível publicamente para utilização através da Internet é, *Anubis* [22], o qual utiliza a técnica de VMI para monitorar as ações de um exemplar de *malware* durante sua execução em um sistema operacional *Windows XP*. A fim de aplicar VMI, *Anubis* foi implementado sobre o emulador *Qemu*, modificado para efetuar a captura de informações de maneira externa ao *guest*. Mais detalhes da implementação de *Anubis* podem ser encontrados em [23].

A principal desvantagem da VMI é que um *malware* pode detectar que está sendo executado em um ambiente emulado/virtual, evitando a análise como um todo ou apresentando um comportamento alternativo ao malicioso. No caso dos emuladores, a detecção pode ser feita de forma muito simples, por exemplo, através da realização de uma instrução no processador que causa um comportamento específico. Um dos modos utilizados para realizar tal verificação é por meio de *bugs* conhecidos em processadores de determinadas arquiteturas que fazem com que certas instruções não se comportem como esperado. Se esta instrução for executada no emulador e este não estiver preparado para apresentar o mesmo comportamento de um processador real, o *malware* irá perceber essa diferença, podendo parar ou modificar a sua execução [24]. A detecção de ambiente virtualizado também é simples, com apenas uma instrução *assembly* que, mesmo executada em um nível de baixo privilégio, retorna informações internas sobre o sistema operacional presente no *guest*. Tais informações identificam o ambiente virtualizado com base nas diferenças entre estes e sistemas reais [25].

Para contornar as técnicas de anti-análise, existem meios de detectar que um *malware* verifica se está em ambiente emulado, ou mesmo de modificar alguns valores presentes no ambiente virtual para tentar disfarçá-lo [26], [27]. Entretanto, o uso destas técnicas muitas vezes é insuficiente e o *malware* ainda pode detectar que está sendo executado em ambiente emulado/virtual.

Um outro sistema utilizado para traçar o comportamento de um *malware* e que se baseia em VMI é *Ether* [28]. Este sistema utiliza VMI para obter as ações realizadas no *guest*, porém, ao contrário de *Anubis*, *Ether* se utiliza de virtualização direta do *hardware*, o que o torna imune às técnicas de anti-análise que verificam se o *hardware* é real ou emulado. A implementação da VMI é feita em uma versão modificada do *Xen hypervisor*, um *software* de virtualização. Uma vantagem de *Ether* sobre *Anubis* diz respeito à análise de

exemplares de *malware* com *packers* que apresentam mau funcionamento em emuladores. Diferentemente de *Anubis*, *Ether* consegue analisar este tipo de *malware* sem qualquer problema em sua execução, dado que o *Xen* utiliza o *hardware* nativo para executar as operações do processador. Entretanto, *Ether* apresenta problemas de desempenho para obter o traço composto pelas *syscalls* que o *malware* realizou. Isso ocorre porque cada chamada de sistema executada pelo programa gera uma *page fault*, a qual é tratada pelo componente de *Ether* responsável pela obtenção do referido traço. Além disso, apesar de ser dito em sua documentação que não é possível detectar sua presença, existem meios de verificá-la, como os descritos em [29]. Nesta referência, são apontados alguns possíveis modos de detectar a presença de *Ether*, como por exemplo através de uma modificação feita pelo sistema de análise que desabilita o bit `TSC` (*Time-Stamp Counter*). Este bit é retornado quando se executa a instrução `CPUID` e serve para indicar quando a instrução *RDTSC* é suportada. Portanto, para detectar a execução em *Ether*, basta que se execute a instrução `CPUID` e se observe o valor retornado no bit `TSC`.

Além dos problemas apresentados com o uso da VMI para captura de informações, há uma outra limitação que diz respeito ao desempenho. Como o componente que obtém as informações fica na camada da VMI, que faz o interfaceamento entre o *guest* e o *host*, os dados capturados são de nível mais baixo, isto é, valores encontrados em registradores da CPU ou endereços de memória. Porém, a análise do comportamento do *malware*, isto é, as modificações feitas no sistema da vítima, requer a obtenção de valores de mais alto nível, como nomes de arquivos criados, registros modificados e processos inicializados. Assim, para acessar tal conteúdo precisa-se interpretar, em tempo de execução, os dados contidos na memória e no processador durante a monitoração do *malware*, o que na maioria das vezes não é uma tarefa fácil e causa uma sobrecarga no processo de análise.

2.2 Hooking

A técnica de *hooking* pode ser definida como um meio de se alterar as requisições e respostas resultantes das interações realizadas em um sistema operacional ou por suas aplicações, através da interceptação das funções ou eventos utilizados [30]. Pode-se categorizar *hooking* como sendo de modo de usuário (*userland hooking*) ou de modo de *kernel* (*kernel hooking*). O que difere estes dois tipos é a extensão da modificação que pode ser feita no sistema e, consequentemente, nas aplicações. *Malware* geralmente utilizam-se de técnicas de *hooking* para capturar ou modificar informações que estejam transitando em uma aplicação ou no sistema operacional. Através disto é possível a ocultação de suas atividades, dificultando assim a sua identificação. Alguns *rootkits* empregam *hooking* para tornar sua presença indetectável ao sistema [21]. Nas seções a seguir, serão detalhadas as diferenças existentes entre *hooking* de nível de usuário e de *kernel*. Serão citados também

exemplos de cada uma das abordagens.

2.2.1 Userland Hooking

Userland hooking ou *hooking* de nível de usuário é uma técnica de interceptação que pode afetar somente programas que executam em nível de usuário, não podendo interferir em qualquer aplicação que opere em um nível mais privilegiado. Mesmo com esta limitação, tal técnica é bastante utilizada por *malware*, dado que sua implementação é mais simples. Embora sua utilização seja frequente, *userland hooking* pode ser facilmente detectado, o que pode levar o programa a desfazer o *hooking* ou não executar a função modificada. Como esse tipo de *hooking* é feito normalmente sobre APIs[1] disponibilizadas pelo sistema operacional, para um programa detectar se a interceptação está sendo feita ou não basta verificar se os endereços das APIs utilizadas por ele estão modificados ou se o endereço é uma instrução de pulo incondicional (JMP), que é uma prática comumente utilizada em *hooking*. Além deste tipo de detecção, existem outras formas que podem ser utilizadas para evitar um possível *hooking*. Uma delas, muito eficaz, é a utilização de funções nativas do sistema operacional, ao invés das APIs fornecidas por ele. Entretanto, empregar este processo requer um maior cuidado na implementação do programa, pois este deverá fornecer um número maior de informações quando for executar cada função, tarefa esta que antes ficava a cargo do sistema operacional. A fim de exemplificar técnicas de *userland hooking* para modificação de APIs, apresenta-se a seguir *IAT hooking*, *Detours* e *inline hooking*.

IAT Hooking

Import Address Table hooking, ou (*IAT hooking*), é uma técnica aplicada para interceptar as funções utilizadas por um determinado programa, antes que este esteja em execução. Para isso, a técnica deve ser empregada sem que o *malware* tenha comprometido o sistema. A *Import Address Table* é uma estrutura presente no cabeçalho de arquivos do tipo PE32 (arquivos executáveis do sistema *Windows*), e é responsável por indicar os endereços das rotinas externas utilizadas por um programa [31]. Esses endereços são definidos no processo de carregamento do programa que antecede a sua execução, quando este está sendo inicializado pelo sistema operacional. Tais endereços são fornecidos por DLLs (*dynamic-link libraries*), pacotes binários que contêm funções e variáveis as quais podem ser utilizadas por outros programas [13]. Quando o programa está na fase de inicialização, o sistema operacional se encarrega de verificar quais DLLs e métodos externos são utilizadas por ele. De posse destas informações, o sistema operacional pode preencher

[1] Application Programming Interfaces

a IAT com os endereços referentes aos métodos usados, de forma que durante a execução o programa carregado consiga invocar corretamente as funções [30].

Para realizar este tipo de *hooking* é necessário modificar a tabela IAT do programa que se deseja monitorar, de forma que os endereços contidos nela sejam de funções que se tem controle. Portanto, para cada função modificada é necessária uma nova função, a qual poderá modificar ou simplesmente monitorar os dados passados pelo programa sob análise para a função original. Além disso, cada função deve invocar a função original para que as ações produzidas por um programa tenham seu efeito consumado no sistema, prosseguindo assim com a execução normal do programa monitorado. Um problema evidente desta abordagem é a necessidade de modificações no programa sob análise, a fim de que seja possível instalar os *hookings* nas *APISs* monitoradas. A detecção deste tipo de ação pode ser feita com um simples teste de integridade no código do programa monitorado. Se isto ocorrer, um *malware* pode identificar que está sendo monitorado, o que pode fazê-lo tomar medidas que inviabilizem sua análise ou que os resultados retornados por esta não correspondam ao fluxo de execução malicioso pretendido originalmente. Outro problema com a abordagem que pode inutilizar a captura ocorre caso o *malware* carregue a DLL que irá utilizar durante a sua execução. Para isso, basta que ele use APIs disponibilizadas pelo *Windows*, as quais possibilitam que a DLL seja utilizada, mesmo sem ser previamente inicializada com o *malware*. Esta prática é bem simples de ser utilizada e desabilita por completo o *IAT hooking*, já que as APIs utilizadas assim pelo *malware* não serão afetadas.

Detours

Uma outra forma de interceptar *APIs* do *Windows* é através do uso de *Detours*, uma biblioteca provida pela própria *Microsoft* para interceptar funções do *Windows* em arquiteturas x86 [32]. Através de sua utilização, é possível realizar modificações no início da função que se deseja interceptar de maneira dinâmica, durante a execução do programa. Esta técnica é implementada através da inserção de uma instrução *assembly* de pulo incondicional (`JMP`) no início da função. Assim, quando tal função for invocada, o *JMP* será executado e irá direcionar o fluxo de execução para uma região sobre a qual se tem controle. Isto possibilita a captura dos dados que estão passando pela função, bem como os valores retornados com sua execução. Sua implementação pode ser feita de maneira simples, visto que o próprio sistema operacional provê suporte para isso, através de um programa que executa em nível de usuário.

Um sistema *open source* para análise dinâmica de *malware* disponibilizado recentemente, chamado *Cuckoobox* [33], aplica a técnica de *Detours* para obter as informações das APIs utilizadas pelo *malware* durante sua execução. Embora o sistema necessite de máquinas virtuais para realizar uma análise, o mecanismo de captura é inserido no *guest* e passa as informações obtidas via protocolo de comunicação para o *host*, diferentemente

de *Anubis*, que requer uma modificação no *software* de emulação para capturar as informações de execução do *malware*. Isto torna a implementação da técnica mais simples, porém, seu custo é que pode-se facilmente verificar o uso de *Detours* através de uma checagem no início da função que se quer realizar: se a instrução inicial for um *JMP*, há a presença de um *Detour*.

Inline Hooking

Inline Hooking é uma outra técnica que pode ser utilizada para redirecionar o fluxo de execução normal de um programa para uma região que se tenha total controle. Em geral, esta técnica é utilizada em *malware* para alterar as APIs do sistema operacional, de forma que as respostas produzidas sejam capturadas ou modificadas. Cabe ressaltar que *inline hooking* pode ser implementada tanto no nível do usuário como no do *kernel*, entretanto, sua forma mais comum aparece em nível de usuário devido à simplicidade da implementação. Seu funcionamento é bem parecido com o do *Detours*, mas em vez de trocar somente a primeira instrução *assembly* da função que se deseja interceptar, é possível alterar uma porção maior de código. Como a parte que desvia a execução do programa para a região de que se tem controle não fica no início do código, a detecção do *hooking* é mais complicada, visto que será necessário inspecionar uma área maior do código que se quer executar em busca de algum desvio de execução.

O sistema de análise de *malware CWSandBox* [34] utiliza a técnica de *inline hooking* para capturar as informações resultantes da execução de um *malware* em um sistema *Windows XP*. O *inline hooking* feito por ele é similar à técnica do *Windows Detour*, onde o começo da função que se deseja interceptar é substituído por um *JMP* [35].

2.2.2 Kernel Hooking

O *kernel hooking*, ou *hooking* em nível de *kernel*, executa em um nível mais privilegiado, utilizando técnicas mais complexas que não são trivialmente detectadas por *malware*. Isto atribui uma vantagem sobre o *userland hooking*, pois torna a sua detecção mais difícil. Porém, na maioria das vezes a detecção de *kernel hooking* pode ser feita por programas que executam em nível privilegiado, como por exemplo os *rootkits* [21]. No caso geral, a análise de *malware* cuja execução ocorre no nível de usuário é mais confiável caso se aplique a técnica de *kernel hooking*, pois o componente responsável pela captura das ações do *malware* está em um nível de privilégio mais elevado, cujo acesso direto não é permitido. Por outro lado, mesmo com a possibilidade de subversão do *hooking*, os *rootkits* (e programas de nível de *kernel* em geral) precisam ser inicializados por programas de nível de usuário. Portanto, todas as ações executadas durante o processo de inicialização, como por exemplo, o carregamento de um *driver*, são capturadas, podendo ao menos

levantar suspeitas sobre um possível comportamento malicioso. Um exemplo de *kernel hooking* comumente utilizado por mecanismos de segurança, como os antivírus, e também por *rootkits* é a técnica de *SSDT hooking*, explicada a seguir. Outro exemplo, explicado adiante, é a técnica de *kernel callbacks*.

SSDT Hooking

O *hooking* da *System Service Dispatch Table*, comumente conhecido como *SSDT hooking*, consiste na modificação de uma estrutura interna presente em sistemas *Windows*, a qual é responsável por armazenar os endereços das *syscalls* do sistema. Esta estrutura é composta basicamente por um vetor de endereços, onde cada índice corresponde a uma das rotinas de chamadas de sistema disponibilizadas pelo sistema operacional, representando uma tabela. Tal tabela reside no *kernel* do sistema operacional e, portanto, programas no nível de usuário não têm acesso a ela. Esta tabela é utilizada pelo sistema operacional quando uma chamada de sistema é requisitada, retornando assim o endereço de memória da função apropriada [14]. Para realizar o *hooking* de SSDT é preciso utilizar um *driver* que opere em nível de *kernel*. Como esse *driver* executa em modo privilegiado, ele pode realizar alterações em outros programas e estruturas internas do sistema presentes no nível de *kernel*. Portanto, o *driver* tem permissão para alterar os endereços contidos na SSDT, trocando-os por valores que indiquem métodos de seu controle. Antes de realizar a troca, os endereços originais precisam ser armazenados para que possam ser utilizados posteriormente, completando assim a requisição feita originalmente. Os endereços alterados irão apontar para funções interceptadas, que serão executadas ao invés das originais. Como se tem o controle destas funções, é possível monitorar as requisições feitas ao sistema e os valores retornados, ou modificar as respostas retornadas pelo sistema operacional ao programa sob análise. Além disso, fica a cargo das funções controladas realizar a chamada às *syscalls* originais, através dos endereços salvos antes das modificações na SSDT, possibilitando que o fluxo de execução original de um *malware* monitorado seja mantido.

Técnicas de *kernel hooking* são muito mais poderosas do que as de *userland* justamente por atuarem em um nível privilegiado, o que lhes permite maior controle sobre os demais programas que executam no sistema operacional. Outra vantagem é que o monitoramento fica transparente para as aplicações de nível de usuário, pois elas não têm acesso às aplicações que executam no nível do *kernel*. Uma desvantagem deste tipo de técnica diz respeito às informações que são extraídas das funções que se pode interceptar. Na abordagem de IAT *hooking*, é possível capturar funções contidas nas *DLLs* utilizadas por um programa, as quais correspondem muitas vezes ao modo utilizado pelo programador para executar a chamada de sistema. Já no caso das chamadas de sistema capturadas através do SSDT *hooking*, a informação não se apresenta de uma forma tão clara. Por exemplo, caso se queira obter o nome de um arquivo utilizado durante uma *syscall*, pode ser ne-

cessário realizar a verificação do conteúdo de outras estruturas utilizadas na invocação da *syscall*, dado que este nome muitas vezes não é passado de forma direta. Neste caso é necessário realizar alguns procedimentos para "traduzir" os dados passados como argumento para a *syscall* de forma a encontrar a informação desejada. Além deste problema, o SSDT *hooking* é dependente da versão do sistema onde ele está sendo aplicado, sendo que para cada versão o SSDT *hooking* deve ser feito de uma maneira diferente. Tal efeito acontece pois podem ocorrer modificações na SSDT entre as versões, fazendo com que endereços antes utilizados para identificar uma *syscall* não sejam os mesmos.

Um sistema de análise de *malware* que faz uso desta técnica é o *JoeBox* [36]. Além de interceptar as chamadas da SSDT ele também realiza um *hooking* de nível de usuário, o que possibilita obter um volume bem maior de informação. Infelizmente só é possível submeter um número limitado de *malware* a este sistema, dado que se trata de um sistema comercial.

2.3 Kernel Callbacks

Callbacks são funções disponibilizadas pelo sistema operacional que notificam uma aplicação sobre determinadas modificações no sistema, como por exemplo, a criação de uma chave de registro ou de um novo arquivo [37]. Estas funções são bem documentadas e, portanto, sua implementação não apresenta incompatibilidades entre as diferentes versões do *Windows*, como ocorre no caso do *SSDT Hooking*, que realiza modificações em estruturas do *kernel* específico de cada versão do sistema operacional. Isto torna possível a monitoração de determinadas ações que ocorrem no sistema de uma forma mais simples e genérica, permitindo a identificação de comportamento possivelmente malicioso. Uma limitação presente nesta abordagem é que ela permite somente a captura das ações realizadas por funções disponibilizadas pelo sistema operacional, o que pode levar à obtenção de um comportamento de execução incompleto. Uma ferramenta que implementa a técnica de *kernel callbacks* é o CaptureBat [38]. Sua utilização requer o carregamento de *drivers* no sistema operacional a ser monitorado, o que possibilita sua aplicação tanto em máquinas virtuais como reais.

A seguir será apresentada uma tabela que resume as técnicas apresentadas até aqui, com suas vantagens, desvantagens e com a relação de sistemas de análise que as utiliza.

Técnica de Análise Dinâmica	**Vantagens**	**Desvantagens**	**Sistema de Análise**
Virtual Machine Introspection	Captura feita fora do ambiente de análise. Análise pode ser feita de forma transparente. Informações de mais baixo nível do ambiente de análise.	Mais complicado obter informações de mais alto nível. O *malware* pode detectar o ambiente onde está sendo feita a análise (emulado/virtual). O ambiente de análise precisa utilizar um ambiente virtualizada/emulada. Dificuldade em obter informações de mais alto nível.	*Anubis*, *Ether*
Userland Hooking	Informação de mais alto nível. Facil implementação.	Pode ser facilmente detectado pelo *malware*. *Malware* pode executar ações de forma direta, evitando o componente de captura.	*CWSandbox*, *Cuckoo-Box*
Kernel Hooking	Componente de captura em nível mais privilegiado. Informação de nível intermediário.	Não é portável entre versões diferentes. O *malware* pode detectar o componente de captura. Componente de captura dentro do ambiente de análise.	BehEMOT, *Joebox*
Kernel Callbacks	Componente de Captura em nível mais privilegiado. Informação de nível intermediário. Portável entre versões diferentes.	Limitação no que se pode capturar. O *malware* pode detectar o componente de captura.	*Capture-BAT*

Tabela 2.1: Técnicas utilizadas na análise dinâmica de *malware* com suas vantages, desvantagens e sistemas de análise disponíveis que as utilizam.

Capítulo 3

Descrição do Sistema

Após o estudo das possíveis técnicas para análise dinâmica, foi possível comparar as vantagens e desvantagens que cada uma apresenta. Esta informação, aliada às características e funcionalidades dos sistemas de análise de *malware* que se utilizam das técnicas apresentadas, tornou possível a escolha de uma abordagem para a implementação do protótipo proposto nesta dissertação. Os requisitos considerados foram a independência de ambiente de execução (virtual, emulado ou real) e a menor possibilidade de subversão ou evasão por parte do *malware* sob análise. Neste capítulo será descrita de forma detalhada a implementação de BehEMOT[1], uma ferramenta para avaliação de comportamento de *malware* através da observação de sua execução, bem como a arquitetura e os componentes do sistema de análise dinâmica que se baseia nela.

3.1 Abordagem utilizada

As abordagens utilizadas em outros sistemas de análise serviram para indicar o rumo a ser tomado na implementação do sistema. A escolha teve como principal fator a desvinculação do sistema a quaisquer tipos de ambientes (virtual/emulado), de forma que fosse possível realizar a análise em qualquer um deles, ou mesmo em uma máquina real. Outro fator determinante é relacionado as possíveis formas de detecção que um exemplar de *malware* pode empregar para verificar que está sendo monitorado. A técnica que apresentou possibilidades triviais de detecção trivial foi descartada, pois o *malware* pode facilmente subverter a análise, escondendo assim o seu comportamento malicioso. Levando em conta os fatores supracitados, escolheu-se a técnica de *SSDT Hooking*, com a finalidade de efetuar a captura das ações realizadas pelo programa malicioso durante sua execução. Dado que a maior parte dos usuários de computador utilizam sistemas

[1] *Behavior Evaluation from Malware Observation Tool*

operacionais *Windows* [39], os programas maliciosos encontrados atualmente são, em sua maioria, direcionados a atacar este sistema. Assim, o sistema de análise dinâmica implementado com BehEMOT foi desenvolvido para examinar arquivos executáveis de sistemas operacionais *Windows*, do tipo PE32 *Executable*. Na instalação do ambiente de análise "vítima", foi utilizado o *Windows XP* com *Service Pack 3*.

Visto que aplicou-se a técnica de SSDT *hooking* para monitorar as atividades do artefato malicioso, BehEMOT não apresenta limitação quanto ao ambiente de análise, podendo ser instalada em máquinas reais, virtuais ou emuladores. Isso dá-se devido a ferramenta ter sido desenvolvida sob a forma de um *driver* que é executado no nível do *kernel*. Portanto, para utilização em qualquer ambiente, basta a instalação do *driver* no sistema operacional do ambiente escolhido, com privilégios de administrador.

Esta técnica faz uso de uma estrutura não documentada do sistema *Windows*, a qual contém os endereços das *syscalls* do sistema operacional. A estrutura, uma tabela, é composta por um vetor de endereços onde cada índice do vetor corresponde ao endereço de uma chamada de sistema específica. Quando um programa de nível de usuário deseja realizar uma *syscall*, o sistema operacional acessa esta tabela para então realizar a função. O SSDT *hooking* modifica esses endereços para os de funções que se tem controle, as quais fazem a captura das informações da chamada de sistema (a chamada e seus argumentos). Os dados obtidos destas funções possibilitam gerar posteriormente um perfil comportamental do *malware*, composto pelas ações executadas no ambiente comprometido.

O tráfego de rede produzido pelo *malware* durante a análise também é capturado, por uma máquina diferente da que realiza a análise. A captura do tráfego de rede através de uma máquina externa a de análise possibilita a observação de ações de rede tais quais varreduras, ataques contra outras máquinas etc, mesmo que o *malware* seja um *rootkit*. Para implementar esse componente de monitoração, foi utilizado um *sniffer* de rede [40]— um programa que funciona anexado a uma determinada interface de rede e captura todo o tráfego que passa através dela. A interface de rede da máquina que serve de base para o sistema de análise conecta-se diretamente à máquina externa, que serve como um *gateway*. Logo, para capturar o tráfego de saída proveniente da máquina de análise, basta anexar o *sniffer* nesse *gateway* e filtrar a comunicação cujo endereço de origem é o ambiente comprometido. Com isso, toda atividade de rede gerada pelo *malware* será monitorada e agregada posteriormente às ações desenvolvidas por ele no sistema.

Todas os resultados obtidos são sumarizados no final do processo de análise em um relatório, no qual são apresentados tanto os dados obtidos pela monitoração do sistema quanto da rede. Para facilitar a análise de alto nível das ações executadas no sistema foi desenvolvido um componente para visualização, cujo objetivo é apresentar as ações de acordo com o instante no qual elas aconteceram.

Para contornar o problema de identificação de ambiente virtual/emulado, optou-se pela

utilização de uma abordagem híbrida, composta por uma máquina real e uma emulada através do *Qemu* [16]. A escolha do *Qemu* baseou-se na praticidade de uso e gerenciamento, e na familiaridade com a ferramenta. Além disso, ela é utilizada em vários sistemas de análise de *malware*, devido ser de código aberto, possibilitando que sejam feitas modificações em suas funcionalidades. A ferramenta *Xen* foi desconsiderada pois os processadores utilizados no protótipo não possuiam a técnologia *Intel VT*, necessária para o correto funcionamento da ferramenta.

O processo de análise obedece a seguinte ordem:

1. Inicialmente é feita a extração do comportamento com BehEMOT instalado em um ambiente emulado;

2. Se ao final desta análise forem encontrados problemas de execução (*crash* no sistema ou relatório vazio), ou se o *malware* estiver ofuscado com algum tipo de *packer* que inviabilize a análise em emuladores, este é enviado para análise em um sistema operacional real, também com BehEMOT.

O controle do processo de análise e as verificações são feitas por um componente do sistema chamado de "analisador", detalhado na Seção 3.3.3. Esta abordagem híbrida, possibilita a análise de exemplares de *malware* que apresentam mecanismos de anti-análise referentes a emuladores, o que não é possível ser feito em sistemas tais como *Anubis* e *CWSandBox*. Por outro lado, como a maioria dos exemplares de *malware* pode executar em emuladores, não é necessário realizar todas as análises em sistema real, caso contrário poderia haver um problema de escalabilidade, pois seria necessário uma quantidade muito grande de máquinas. Nas seções que seguem, serão apresentados maiores detalhes sobre a implementação de cada componente de BehEMOT.

3.2 Arquitetura do sistema

BehEMOT foi projetado para integrar um sistema de análise dinâmica de *malware* e consiste de três componentes principais—*driver*, controlador e analisador—e um auxiliar que auxilia na visualização das ações maliciosas. Os componentes principais são necessários para capturar as ações maliciosas e, portanto, devem estar presentes em todas as máquinas onde são feitas as análises. Já o analisador é responsável por capturar o tráfego de rede gerado durante o período em que o *malware* permaneceu em execução, decidir se este deve ser analisado em ambiente real ou emulado e gerar um relatório que sumariza as informações de rede e de ações realizadas no ambiente. Esse componente situa-se fora do sistema de análise, não estando sujeito às alterações que porventura um *malware* realize

durante sua execução. Tal isolamento funciona como um nível de privilégio adicional, invisível ao ambiente de análise. Com isso, fica difícil para o *malware* subverter o resultado da análise, visto que o componente que o gera está em um nível ainda mais privilegiado do que o nível do *kernel* do sistema comprometido. O componente auxiliar, chamado de "visualizador", tem por função facilitar o entendimento da cadeia de atividades desenvolvidas pelo *malware* no ambiente de análise. Por ser um componente adicional, o mesmo não tem nenhuma relação com o procedimento de análise dinâmica e geração do relatório. Logo, o componente visualizador não faz parte da arquitetura do sistema de análise, mostrada a seguir.

Na figura 3.1, pode-se observar como os componentes interagem entre si durante o processo de análise do programa malicioso. A parte demarcada como análise comportamental contém o ambiente de análise, que pode ser real ou virtual. Como descrito anteriormente, os componentes "*driver*" e "controlador" permanecem dentro do ambiente de análise.

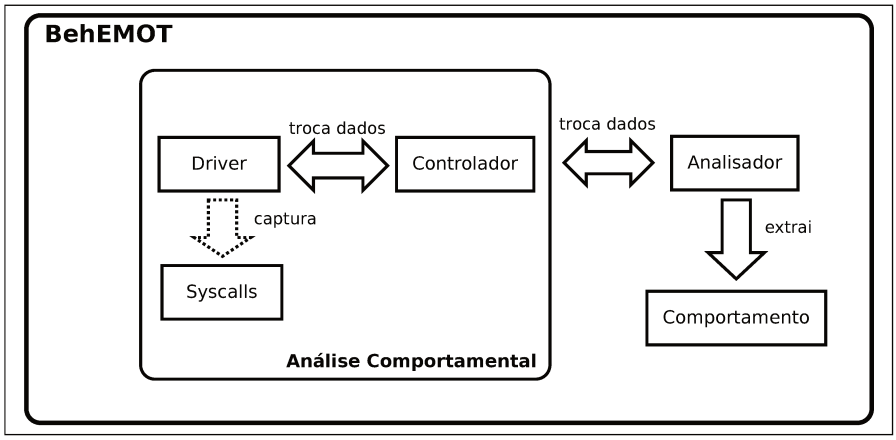

Figura 3.1: BehEMOT e seus componentes

O sistema completo é formado por um *pool* de máquinas emuladas com o *Qemu*, um sistema operacional real (não emulado nem virtualizado), um *firewall* para captura do tráfego de rede, isolamento dos sistemas e contenção de ataques lançados, e um banco de dados para armazenamento de exemplares de *malware* e relatórios de análise. O *firewall* foi configurado de forma que o tráfego de saída do *malware* seja limitado, evitando assim que o mesmo lance ataques a outras máquinas na Internet. O tráfego de entrada é permitido devido aos exemplares de *malware* que requerem o *download* de outros componentes, como os da classe *Downloader* [41], que vêm inicialmente na forma de um programa cujo único propósito é obter o real arquivo malicioso via rede. Através do *pool* de máquinas emuladas, busca-se uma melhor eficiência nas análises, visto que é possível executá várias delas de

forma paralela. A máquina real pode representar um gargalo para o sistema, caso um grande número de exemplares necessitem ser analisados por ela. Porém, tal situação só iria ocorrer caso a maior parte dos exemplares que fossem ser analisados apresentassem mecanismos de detecção de emuladores, o que não acontece na prática [42].

3.2.1 Configuração dos sistemas de análise

As máquinas emuladas dispõem, cada qual, de instalações padrão do *Windows XP SP3*, com a adição apenas dos componentes do controlador e do *driver*. Procurou-se deixar o ambiente o mais próximo possível de uma instalação vista em máquinas de usuários comuns, sem adição de outras ferramentas de monitoração, de modo a evitar que o artefato descubra que se trata de um ambiente de análise. Para otimizar o tempo gasto na inicialização do sistema e manter a integridade do ambiente de análise, foram feitos *snapshots*[2] no estado pronto para monitoração, O *Qemu* dispõe desta funcionalidade, que pode ser utilizada a qualquer momento, mesmo com o sistema em execução. Assim, é possível armazenar um *snapshot* do sistema com os componentes (controlador e *driver*) instalados e prontos para operar, e para o qual o estado do sistema será revertido sempre que uma nova análise for realizada. Ao final da análise, o sistema comprometido por um *malware* é descartado, evitando assim que as próximas análises apresentem resultados não confiáveis e mantendo e estado do ambiente íntegro. Além de evitar um desvio nos resultados finais, o uso de *snapshots* agiliza o processo de análise, pois o tempo que seria gasto com a inicialização do sistema é eliminado, visto que ele volta em um estado pronto para executar o *malware*.

O mesmo processo se aplica para as máquinas reais, com o diferencial que estas possuem duas partições. Em uma delas está o sistema de análise, com o *Windows* instalado de forma padrão, enquanto que na outra está um sistema operacional *Linux*. Este último é necessário para retornar o sistema comprometido pela execução do *malware* para um estado íntegro, anterior a sua execução. Para realizar esta restauração foi utilizada à ferramenta *partimage*, um programa *Open source* que guarda o estado de uma determinada partição em um arquivo para posterior recuperação [44]. Basicamente, o programa copia a partição desejada para um arquivo e, quando há a necessidade de restaurá-la, o programa lê esse arquivo e sobrescreve a partição de destino com os dados contidos nele. É possível comprimir o tamanho do arquivo salvo com os dados da partição, de forma que seja ocupado menos espaço. Tanto o procedimento de armazenamento do estado da partição quanto a sua recuperação precisam ser feitos a partir da partição que contém o sistema *Linux*. A seleção de qual sistema será iniciado é feita pelo *bootloader* durante

[2]trata-se de um termo técnico utilizado para descrever a habilidade de armazenar o estado de um dispositivo, o qual poderá ser restaurado posteriormente [43].

o processo de inicialização, pois este é o primeiro *software* carregado na máquina, sendo responsável por passar o controle para o sistema escolhido [45].

Foi utilizado o *grub* como *bootloader*, por ser de fácil manuseio e possibilitar a modificação de qual sistema será iniciado de forma bem simples. Os passos executados durante uma restauração do sistema de análise que utiliza máquina real são os seguintes:

- O sistema de análise está em estado pronto, com o controlador, no Windows, aguardando um exemplar de *malware*.
- O analisador envia um *malware* para o controlador da máquina, que o executa e monitora suas ações.
- Após a infecção, o analisador envia um comando para reiniciar o ambiente Windows.
- Na hora de escolher qual partição será iniciada, o *bootloader* irá iniciar o sistema *Linux* para que a restauração seja feita. Para isso, o *bootloader* é configurado para escolher esse sistema por padrão.
- Quando o sistema *Linux* carrega, ele executa um *script* que configura o *bootloader* para iniciar o sistema *Windows* somente na próxima vez que ele for executado, restaura a imagem integra do sistema de análise e ao fim disso reinicia o sistema.
- Quando o *bootloader* for executado, ele escolhe o sistema *Windows* deixando o sistema pronto para receber uma nova análise, finalizando o ciclo

O procedimento completo de restauração, contando o tempo necessário para iniciar os sistemas *Windows* e *Linux* demora cerca de 2 minutos, para um sistema *Windows* com 4GB de espaço reservado. Como a rotina de restauração configura o *bootloader* para iniciar o sistema *Windows* somente na próxima vez em que ele for executado, fica descartada a possibilidade de o sistema de análise reiniciar em um estado comprometido.

3.3 Componentes do sistema

Nesta seção serão detalhados os componentes de BehEMOT, bem como as informações sobre sua implementação. Como mencionado, alguns deles estarão presentes internamente ao ambiente de análise, os quais possuem as configurações previamente descritas.

3.3.1 Driver

No início da era dos computadores pessoais, quando os processadores não eram tão poderosos como atualmente e disponibilizavam apenas 640 KB de memória, existia apenas

um modo de operação possível, o *real mode*, no qual todas as aplicações executavam em um só nível de privilégio. Pouco de depois do surgimento dos processadores 386, que permitiam o acesso a 4GB de memória virtual, a *Microsoft* disponibilizou o *Windows* 3.0. Nele, existiam dois modos de privilégio de operação para os programas, o *user mode*, cujos privilégios são limitados e abrangem os programas mais comuns (ex., editores de texto), e o *kernel mode*, no qual são executados os programas que precisam de um nível privilegiado para obter acesso a qualquer recurso do sistema [46].

Esse modelo de divisão de privilégios foi mantido nos sistemas *Windows*, sendo que nas versões de *64 bits* existe um controle mais rígido sobre as aplicações que possuem permissão para executar em *kernel mode*. Quando o usuário deseja executar uma nova aplicação com privilégio de *kernel* é preciso que esta esteja assinada digitalmente por uma autoridade certificadora. Caso contrário, faz-se necessário desabilitar uma restrição do sistema durante o *boot*, permitindo assim que aplicações não assinadas sejam carregadas [47].

Para capturar as ações maliciosas efetuadas no ambiente de análise, foi preciso desenvolver um *driver* cuja operação ocorresse em *kernel mode*, obtendo assim, o acesso a qualquer informação gerada durante a execução do programa monitorada. Caso a captura fosse executada por um programa de *user mode*, o *malware* poderia empregar técnicas simples para detectar que estava sendo monitorado, o que resultaria na evasão ou subversão da análise de forma que nenhuma informação fosse capturada durante a sua execução. O *driver* utilizado em BehEMOT foi desenvolvido em linguagem C e compilado com as ferramentas do WDK (*Windows Driver Kit*), disponibilizadas pela *Microsoft* gratuitamente.

Foi utilizada a técnica de *SSDT Hooking*, conforme mencionado anteriormente, com a finalidade de se realizar a captura das informações relativas à execução de um dado *malware*. Como existe um conjunto de 391 chamadas de sistema presentes na SSDT [14], escolheu-se aquelas cujas ações são representativas para a segurança do sistema, quando executadas por um exemplar de *malware*. A escolha foi baseada em um estudo que analisou um número significativo de programas maliciosos e possibilitou a identificação de tais ações [42]. Além disso, levou-se em conta também alguns métodos comumente utilizados na identificação de *malware*, como por exemplo, a criação de um *mutex* característico ou a escrita em determinados registros. Normalmente, *mutexes* são estruturas disponibilizadas nos sistemas *Windows* para garantir o acesso exclusivo a um determinado recurso. Programas maliciosos utilizam-se disso para descobrir se existem outras instâncias de *malware* em execução na máquina. Tendo como exemplo o *Conficker*, um *malware* recente que infectou um grande número de máquinas [48], nota-se que o mesmo pode ser identificado pelos *mutexes* que cria no sistema durante sua execução [49]. O processamento de todas as chamadas de sistema capturadas torna possível a obtenção de informações referentes às operações em arquivos, chaves de registro, processos, memória, rede e *mutexes*. Para

cada um desses tipos de operação, é necessário implementar uma função que será responsável por obter a informação necessária e executar a *syscall* original, evitando assim problemas na execução do sistema operacional. Para evitar a obtenção de informações que não possuem nenhuma relação com o programa analisado, são capturados somente os dados produzidos pelo processo principal do *malware* e de seus descendentes, ou seja, processos criados por ele ou algum de seus filhos. A identificação é feita pelo *Process Identifier* (PID) do processo, o qual é obtido no momento de sua criação para execução. Portanto, somente serão obtidas as informações geradas por processos que tenham seu PID marcado para monitoração. Esses valores estão armazenados em uma fila, a qual é verificada sempre que uma *syscall* monitorada é executada.

As informações obtidas das *syscalls* executadas pelo *malware* são armazenadas em uma fila do tipo *First In First Out* (FIFO), denominada fila de registro de atividades, na qual a retirada é feita de acordo com a ordem de chegada, ou seja, a informação que chegou primeiro será retirada primeiro. Deste modo, quando o programa que faz a leitura destes dados for lê-los, ele terá os ações na ordem que elas ocorreram. Para evitar que a fila consuma muita memória caso não haja retiradas de informações, existe um temporizador que, após passado um determinado período de tempo sem que ocorra uma leitura, elimina todos os registros da fila. O tempo utilizado é calculado de forma que a fila não seja esvaziada caso as leituras estejam ocorrendo de modo correto. Os registros contidos na fila seguem um padrão, para facilitar a leitura dos mesmos. Neles, encontra-se o *timestamp*, ou seja, a marca de tempo na qual ocorreu o evento, o nome do processo que executou a ação, o tipo de ação que foi efetuada e o alvo, que caracteriza o objeto de destino que se deseja alcançar com a ação. A Figura 3.2 mostra um registro e seus campos. Em "Executor da Ação" estará o nome do programa a partir do qual se origina a ação. Esta informação é obtida através de métodos disponibilizados pelo sistema operacional, os quais retornam o nome do processo que está executando a chamada de sistema em questão. Neste campo só estão presentes programas cujo PID foi marcado para monitoração. No campo "Tipos de Ação", podem ser encontrados os seguintes valores:

- WriteFile, ReadFile, DeleteFile, CreateFile, que são operações em arquivo. O alvo sempre indica o nome do arquivo no qual que se deseja realizar ação.
- CreateKey, SetValueKey, QueryValueKey, DeleteKey, que representam as operações no registro. O alvo sempre indica o nome do registro no qual que se deseja realizar ação.
- CreateMutex, QueryMutex, OpenMutex, ReleaseMutex, que são as operações em *mutexes*. O alvo sempre indica o nome do *mutex* no qual que se deseja realizar ação.

- CreateProcess, TerminateProcess, OpenProcess, representando as operações em processos. O alvo sempre indica o nome do processo no qual que se deseja realizar ação.
- WriteMemory, que é a operação de escrita em memória. O alvo sempre indica o nome do processo no qual que se deseja realizar ação.
- ConnectNet, SendNet, ReceiveNet, DisconnectNet que são as operações de rede. O alvo sempre indica o endereço IP e a porta destinos, separados por ":".

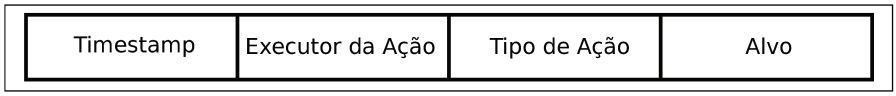

Figura 3.2: Estrutura que armazena as ações executadas no sistema capturadas por BehEMOT.

Estas ações são determinadas através das chamadas de sistema realizadas. Por exemplo, a chamada `ZwWriteFile`, que é utilizada para escrever dados em um arquivo, implica no tipo de ação correspondente `WriteFile`, que indica que uma ação de escrita está sendo realizada em um determinado alvo. Os valores presentes no campo "Alvo" são cadeias de caracteres que indicam arquivos, nomes de registros, de processos e de *mutexes*. Normalmente eles ficam localizados em uma estrutura chamada `OBJECT_ATTRIBUTES`, parâmetro este presente em quase todas as chamadas de sistema modificadas. Dentro desta estrutura está localizado o campo `ObjectName`, que contém uma *string unicode* com a informação desejada. Caso a *syscall* não tenha esse parâmetro, mas tenha um parâmetro `HANDLE`, torna-se possível obter a estrutura `OBJECT_ATTRIBUTES`. Nos sistemas *Windows*, é comum o uso de objetos, tipos de dados abstratos que podem ser instanciados de várias formas. Cada instância tem um tipo bem definido pelo sistema, assim como as operações que são possíveis nele. Por exemplo, há o objeto *Process*, o qual é definido por um processo no sistema operacional. A referência desses objetos é feita através da estrutura `HANDLE` [13].

Certos tipos de *malware* têm sua execução inicializada no sistema na forma de um serviço. Serviços de sistemas *Windows* são similares aos *daemons* presentes em sistemas *Linux*, isto é, são programas que podem inicializar juntamente com o sistema e não precisam de interação com o usuário. Como exemplo, pode-se citar um servidor *Web*, que funciona independente de haver ou não um usuário autenticado no sistema [13]. Para instalar um serviço, é necessário o uso de uma API `CreateService`, a qual se encarrega de realizar todos os passos necessários para a correta instalação. Durante esta etapa é feita a comunicação com o processo em execução `services.exe`, que é responsável por controlar os serviços do sistema. Este é iniciado com o sistema operacional, e é responsável por

carregar os outros serviços marcados para iniciar da mesma maneira. A comunicação da API `CreateService` com o processo `services.exe` é feita através de LPC (*Local Procedure Calls*), um método de comunicação entre processos usados nos sistemas *Windows*. Nesta comunicação, é necessário que haja uma conexão com uma determinada porta, a qual deve estar previamente disponível no sistema. Estabelecida esta conexão, é possível que um programa se comunique com o outro através de mensagens, as quais podem conter instruções que devem ser executadas em outro processo. Para estabelecer esta conexão utiliza-se a chamada de sistema `NtConnectPort`, na qual um dos parâmetros informados é o nome da porta a que se quer conectar. Sabendo de antemão o nome da porta que identifica o processo `services.exe`, é possível identificar quando um dado processo deseja iniciar um serviço. Portanto, quando um *malware* tenta inicializar-se desta forma, pode-se capturar as ações executadas por ele sem qualquer perda de informações, sendo necessário apenas que o processo iniciado pelo `services.exe` correspondente ao *malware* seja marcado para monitoração.

Outra ação muito comum utilizada por programas maliciosos é a escrita na memória de processos de outros programas, por exemplo, com o emprego de *DLL Injection*. O ataque de *DLL Injection* é muito comum entre *malware*. Nela, uma DLL que contém o código com ações maliciosas é carregada em outro processo, o qual passará a realizar ações maliciosas. Geralmente, como o comportamento malicioso estará sendo executado por outro processo, o do *malware* é finalizado. Com isso, o *malware* transfere o comportamento malicioso para outro processo em execução, transferindo-o de seu próprio processo. Caso o sistema de análise utilizado não preveja esta possibilidade, o comportamento observado poderá ser incompleto. Para cobrir este caso, foi modificada a chamada de sistema utilizada na escrita de memória de outros processos, de forma que se algum processo monitorado executá-la, o processo alvo (caracterizado pelo processo onde se quer escrever), também será marcado para monitoração. A partir disso, qualquer ação executada pelo processo modificado será também monitorada, viabilizando a captura de todo o comportamento malicioso existente.

Para ilustrar de forma mais clara o processo de execução de uma chamada de sistema no ambiente de análise, é utilizado o fluxograma descrito na Figura 3.3. Percebe-se que, caso o processo que executa uma dada *syscall* não esteja marcado para monitoração, a chamada de sistema original é executada imediatamente, evitando assim um *overhead* desnecessário. Caso contrário, são obtidos os parâmetros necessários para preencher a estrutura de dados que caracteriza uma ação efetuada pelo *malware* no sistema.

Outra funcionalidade provida pelo *driver* de BehEMOT abrange a leitura e escrita de dados nele feitas por programas de *userland*. Para tratar disso, devem ser implementadas funções, as quais serão responsáveis por responder a estas requisições.

Quando um programa qualquer deseja realizar uma leitura no *driver*, ele deve utilizar

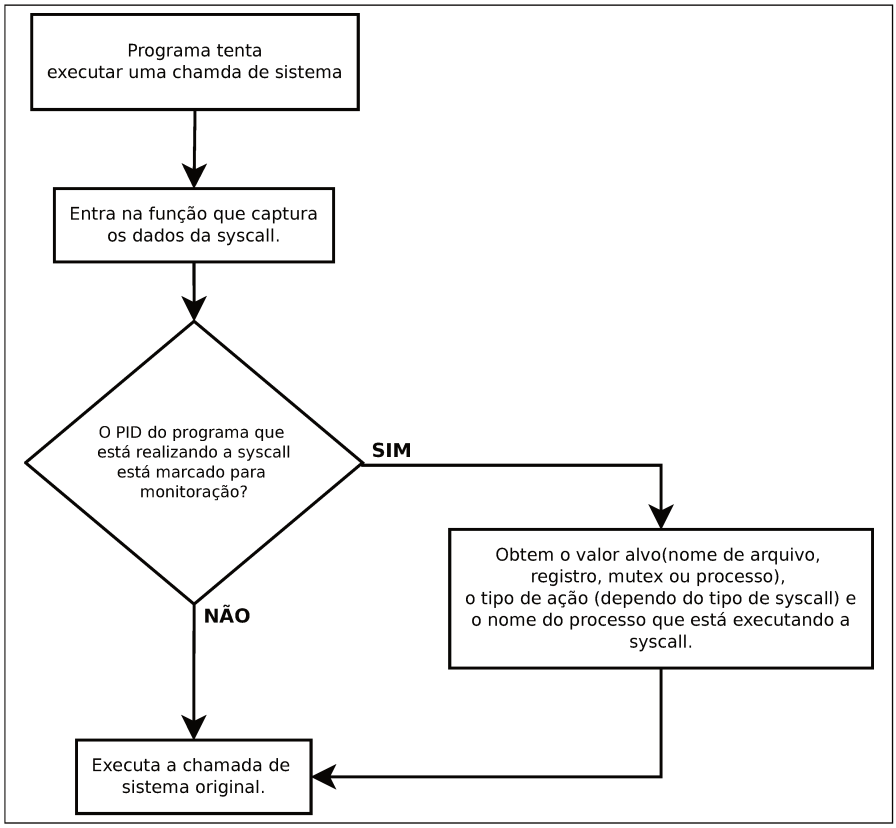

Figura 3.3: Fluxo de execução de uma chamada de sistema em BehEMOT.

uma chamada de sistema específica, passando o *driver* como parâmetro, juntamente com um *buffer* pré-alocado onde serão inseridas as informações lidas. Para transferir dados entre o *driver* e uma aplicação de *userland* é necessário o uso de *I/O Request Packet* (IRP), uma estrutura de dados utilizada na comunicação entre *drivers* e programas disponibilizada pelo sistema. Esta estrutura é utilizada sempre que se deseja ler ou escrever informações em um *driver*, ficando sob responsabilidade deste o preenchimento dela, no caso de uma operação de leitura, ou sua leitura, no caso de uma operação de escrita. As operações de leitura e escrita são tratadas por funções do *driver*, as quais são acionadas sempre que um programa de *userland* desejar interagir com ele. No *driver* de BehEMOT, existe uma função para cada uma destas operações (leitura e escrita).

A função que trata a escrita, ou seja, que recebe dados, obtém o valor de um PID (no caso, do processo do *malware* que será analisado) e o marca para monitoração. Com esta finalidade, a função lê o valor e o insere em uma fila que contém todos os PIDs marcados

para monitoração no sistema de análise. Para a leitura dos dados, ou seja, quando um programa deseja receber informações do *driver*, a função fica responsável por preencher a IRP que é lida pelo programa. Tal função obtém um dos registros da fila de registro de atividades, a qual armazena as ações obtidas dos processos cujos PID estão marcados para monitoração, coloca o registro no *buffer* previamente alocado pelo programa de nível de usuário, e informa o sistema, o qual se encarrega de repassar a IRP para o referido programa. Na implementação corrente somente um registro é lido por vez; a leitura de várias informações no *buffer* é sugerida como expansão futura.

3.3.2 Controlador

Para fazer a interface entre o *driver* e o ambiente externo da análise, bem como controlar o início e fim da análise, é utilizado um programa que executa em nível de usuário (*Ring* 3), o controlador, que realiza as seguintes atividades:

- Carregar o *driver* responsável pela captura das informações.
- Inicializar o processo do *malware* que se quer monitorar.
- Obter as informações capturadas pelo *driver* e enviá-las para o analisador de BehEMOT.

Um *driver* pode ser carregado em sistemas *Windows* de duas formas: uma que segue os métodos documentados e é mais segura, e aquela que segue métodos não documentados e levanta menos suspeitas sobre a presença do *driver* no sistema. É possível esconder as evidências da execução de um *driver* de modo que seu rastreio seja somente possível caso se submeta a máquina a uma perícia forense [14]. A maneira segura e documentada de carregar o *driver* é através do *Service Control Manager* (SCM), que possibilita a inicialização e término de serviços no sistema. Entende-se por serviços os programas que executam em *background* e normalmente ficam à espera de um determinado evento, como um servidor [13].

O SCM é um programa de linha de comando que é executado durante o processo de inicialização do sistema. Ele é responsável pelo controle dos serviços que funcionam no sistema provendo interfaces para a inicialização e término dos mesmos. Durante o processo de *boot*, o SCM cria uma base de dados com os serviços presentes no sistema realizando, para isso, a leitura do registro `HKLM\SYSTEM\CurrentControlSet\Services` e cria uma entrada na base para cada uma das chaves encontradas. Após isso, os *drivers* e serviços que estão assinalados para inicializar durante o *start up* do sistema são executados. Com o uso do SCM, fica assegurado que o *driver* carregado através dele não corre o perigo de ser paginado da memória do sistema, ou seja, que seu código seja desalocado da memória

e transferido para o disco. Caso isso ocorresse e uma *syscall* estivesse sendo interceptada, o sistema iria produzir uma exceção não tratada pelo *kernel*, conhecida como *Blue Screen of Death* (BSOD) [21]. Devido ao sistema ser restaurado a um estado íntegro após a análise ser completada, o *driver* estará sempre carregado.

Após o *driver* ser carregado, o controlador de BehEMOT inicializa o processo do *malware* em estado suspenso. Desta forma, é possível obter o PID do *malware* que será enviado ao *driver*, o qual irá marcá-lo para monitoração. Usando esta abordagem, não será perdida nenhuma informação referente à execução do *malware*, pois pode-se marcar o processo antes mesmo deste entrar em execução. Para transferir este valor para o *driver*, o controlador envia um *I/O Request Packet* (IRP)—uma estrutura de dados utilizada na comunicação com *drivers* do sistema—com o valor do PID. As ações de leitura e escrita são tratadas por funções do *driver*, as quais são acionadas sempre que o controlador precisar interagir com ele. Como descrito na seção anterior, o *driver* recebe o valor do PID referente ao processo malicioso passado pelo controlador e indica que ele deve ser monitorado. Após isso, o controlador ajusta o tempo máximo da análise em quatro minutos, de forma que ao término deste período, a captura de dados do *driver* pare, o arquivo que contém as informações obtidas seja encerrado e enviado à máquina fora do ambiente de análise e, no caso do ambiente real, o computador seja reiniciado. Se for o ambiente emulado, o analisador (Seção 3.3.3) fica responsável por finalizar o processo do emulador Qemu. Configurado o tempo de análise, o controlador coloca o processo do *malware* em execução e em seguida, entra em *loop*, realizando a leitura síncrona, das informações capturadas pelo *driver*. Como o *driver* armazena os dados capturados em uma fila FIFO, mesmo que o processo do *malware* seja iniciado antes da leitura, nenhuma ação será perdida dado que elas estão armazenadas nesta fila. Todo o conteúdo lido do *driver* pelo controlador é enviado através da rede para o componente do analisador. Esta comunicação é filtrada do tráfego de rede capturado para não gerar ruído no comportamento malicioso. O envio das informações é feito logo após a leitura de um registro do *driver*.

Quando o tempo de análise termina a comunicação é encerrada, e é enviada uma notificação ao analisador. O componente analisador de BehEMOT fica responsável pelo tratamento dos dados e agregação do tráfego de rede capturado ao comportamento obtido no sistema operacional. Terminada a captura, os procedimentos previamente descritos para finalizar o processo de análise são executados e o ambiente é restaurado a um estado íntegro.

3.3.3 Analisador

O componente responsável por decidir em qual ambiente (real ou emulado) o *malware* será executado, capturar o tráfego de rede gerado durante a análise, apresentar as ativi-

dades executadas pelo programa malicioso no sistema-alvo juntamente e filtrar as ações efetuadas de forma a exibir somente o comportamento malicioso relevante é chamado de "analisador". Este componente é localizado externamente ao ambiente de análise e, portanto, não sofre os efeitos produzidos pelo comportamento malicioso do *malware*. Como a comunicação entre os componentes internos ao ambiente de análise e o analisador é feita através de uma aplicação cliente-servidor, o *firewall* do sistema no qual o analisador executa é configurado de forma a evitar qualquer acesso indevido por parte do *malware*. Para isso, são aceitos apenas dados originados do ambiente de análise que tenham uma porta origem conhecida, utilizada pelo controlador de BehEMOT para enviar os dados capturados durante a execução do *malware*.

A escolha do ambiente de análise (real ou emulado) é feita baseada em dois fatores:

- (i) a presença de *packer* que impossibilite a análise em ambiente emulado e,
- (ii) o resultado obtido após uma análise em ambiente emulado ser vazio ou apresentar algum problema identificado.

Para se tentar identificar o tipo de *packer* presente no exemplar sob análise, utiliza-se a ferramenta PEiD[3] [50], a qual realiza uma consulta em um arquivo que contém assinaturas conhecidas de *packers* e verifica se o *malware* em questão possui uma ou mais delas. O arquivo de assinaturas é composto por um conjunto constantemente atualizado e diversificado de identificadores, sendo que os *packer* conhecidos que são utilizados para inviabilizar a análise em ambientes emulados também estão presentes neste. De acordo com [24], exemplares de *malware* ofuscados com *tElock!*, certas versões de *Armadillo* e *Asprotect* não são executados corretamente em ambientes emulados com Qemu. Portanto, quando um artefato é submetido para BehEMOT, seu componente analisador verifica, através da aplicação do PEiD, se há presença de algum desses *packers* supracitados. Caso haja a identificação, o analisador envia o exemplar para o sistema de análise que se utiliza de máquina real. Caso contrário, a análise é realizada no ambiente de análise emulado.

O segundo fator que indica o ambiente no qual o *malware* deve ser executado baseia-se no resultado obtido de uma análise prévia em ambiente emulado. Após o término da análise, o analisador primeiramente verifica se esta terminou com sucesso, isto é, sem que erros identificáveis tenham ocorrido. Para isso, durante a verificação, o analisador observa se o término do Qemu foi normal ou se apresentou algum problema durante sua execução. Certos tipos de *malware* causam problemas no *software* de emulação que, em consequência destes, termina de forma inesperada. Caso haja a confirmação de algum problema, a execução em ambiente real é necessária, ficando a cargo do analisador enviar o exemplar para uma nova análise nesse ambiente. Um outro problema que pode ocorrer durante a

[3]Portable Executable Identifier

análise no ambiente emulado tem relação com o comportamento apresentado pelo *malware*. Determinados tipos de *malware* podem, ao invés de realizar uma ação que causa uma exceção inesperada no Qemu, simplesmente terminar sua execução, ocultando assim seu comportamento malicioso. Se isto acontecer, o resultado da análise deste exemplar é vazio, fazendo com que o analisador envie o *malware* para uma nova análise, desta vez no ambiente real, na tentativa de obter seu comportamento malicioso.

A obtenção do tráfego de rede é realizada através da ferramenta *tcpdump*, um capturador de pacotes cuja versatilidade torna possível a criação de filtros via linha de comando, além de permitir o fácil controle da ferramenta por meio de *scripts* em *shell* de *linux* [51]. É possível usar esses filtros do *tcpdump* com a finalidade de se capturar somente a parte do tráfego necessária, isto é, aquela referente às ações maliciosas, desconsiderando-se tanto eventuais pacotes gerados pelo funcionamento normal do sistema operacional e que podem ser observados na rede, quanto a comunicação produzida entre os componentes controlador e analisador de BehEMOT. Além disso, a filtragem permite a captura do tráfego gerado pelo sistema de análise responsável pela execução do programa malicioso em questão, separando-o do tráfego gerado por outros sistemas de análise que estejam em execução no mesmo momento. Tal precaução é necessária devido à interface de rede na qual é capturado o tráfego ser compartilhada entre as máquinas de análise, seja do ambiente emulado ou do real. Com esta abordagem, é possível que todo o tráfego de rede gerado pelo ambiente de análise seja capturado, mesmo que eventualmente não seja obtido nenhuma ação da execução do *malware*.

Além das atribuições descritas anteriormente, o analisador deve filtrar as ações efetuadas durante a execução do *malware* no sistema operacional do ambiente de análise. O objetivo desse processo é extrair somente aspectos do comportamento malicioso do programa analisado, descartando eventuais ruídos provenientes da interação normal com o sistema operacional ou ações que não agreguem informações úteis. Nesta filtragem são desconsideradas, por exemplo, as ações de abertura e leitura de arquivo, leitura de alguns registros e atividades repetidas. As aberturas de arquivo são desconsideradas, pois só passam a informação de que um processo solicitou a obtenção de um *handle* para um determinado arquivo. Se qualquer modificação for feita no arquivo, esta será reportada através das atividades de escrita capturadas, tornando a abertura dispensável. Já as ações de leitura de arquivo ou registro são descartadas por não realizarem mudanças no sistema. Estas atividades são utilizadas para verificar valores do sistema, não realizando portanto qualquer modificação no ambiente. Ações repetidas também são filtradas de forma que só esteja presente no relatório uma única ocorrência da ação. Para ilustrar o ruído presente nas ações repetidas, toma-se como exemplo o caso de uma escrita em arquivo. Um programa sob análise pode realizar esta operação centenas de vezes, dependendo da quantidade de informação que seja armazenada no arquivo aberto para escrita.

No comportamento não filtrado, estariam presentes várias linhas repetidas provenientes das diversas escritas, poluindo o resultado da análise e dificultando o seu entendimento e sua comparação com outros relatórios de execução. Aplicando-se o processo de filtragem, será obtida apenas a primeira escrita feita pelo *malware* em um determinado arquivo, facilitando assim o entendimento do comportamento final apresentado.

3.3.4 Visualizador

O comportamento obtido através do sistema de análise toma a forma de um arquivo textual que agrega todas as ações decorrentes da execução do *malware* no ambiente de análise. Como tal arquivo tende a ser extenso, dependendo do tanto de atividades realizadas pelo programa malicioso, torna-se complicada a compreensão geral do comportamento apresentado por um analista humano. Visando facilitar o processo de análise de resultados, criou-se o componente visualizador, o qual apresenta de forma gráfica o comportamento obtido da execução de um dado exemplar de *malware*. Esta representação gráfica é gerada a partir das atividades provenientes do arquivo textual, através do encadeamento destas de acordo com o processo que as executou. Cada ação é representada por um retângulo e é ligada por setas que indicam a ordem de execução. A disposição das ações foi organizada de forma que aquelas que ocorrem em períodos de tempo (*timestamp*) diferentes fiquem em linhas separadas. Conforme o *timestamp* aumenta, as linhas crescem no sentido vertical, e as mais antigas são posicionadas acima das mais recentes. Além disso, quando ocorre uma ação de criação de um novo processo, esta é marcada em vermelho, a fim de identificar um outro fluxo de execução. Esse fluxo paralelo, proveniente de um processo-filho, é apresentado em uma nova janela e mostra apenas o comportamento do novo processo. A Figura 3.4 mostra o fluxo de execução principal de um exemplar de *malware* sob análise, até o momento da criação de um novo processo.

Cabe ressaltar que as atividades realizadas por este *malware*, se existentes, continuam a ser apresentadas neste mesmo fluxo, como pode ser observado na mesma figura, após o retângulo vermelho. O fluxo de execução do novo processo, criado durante a execução do exemplar exemplificado na figura anterior, pode ser observado, separadamente, na Figura 3.5.

É possível notar, através das figuras mostradas acima, certas ações que são representadas em uma mesma linha. Isto ocorre devido ao *timestamp* obtido ser o mesmo, dado que a diferença de tempo entre elas é tão pequena que não pode ser capturada. Entretanto, como as ações estão armazenadas em uma fila FIFO, aquelas que ocorrem primeiro são representadas mais à esquerda. Portanto, mesmo que o *timestamp* obtido seja o mesmo, a ordem de execução é mantida. Além disso, cada um dos processos presentes no comportamento geral tem suas atividades apresentadas em uma janela individual. Isto possibilita

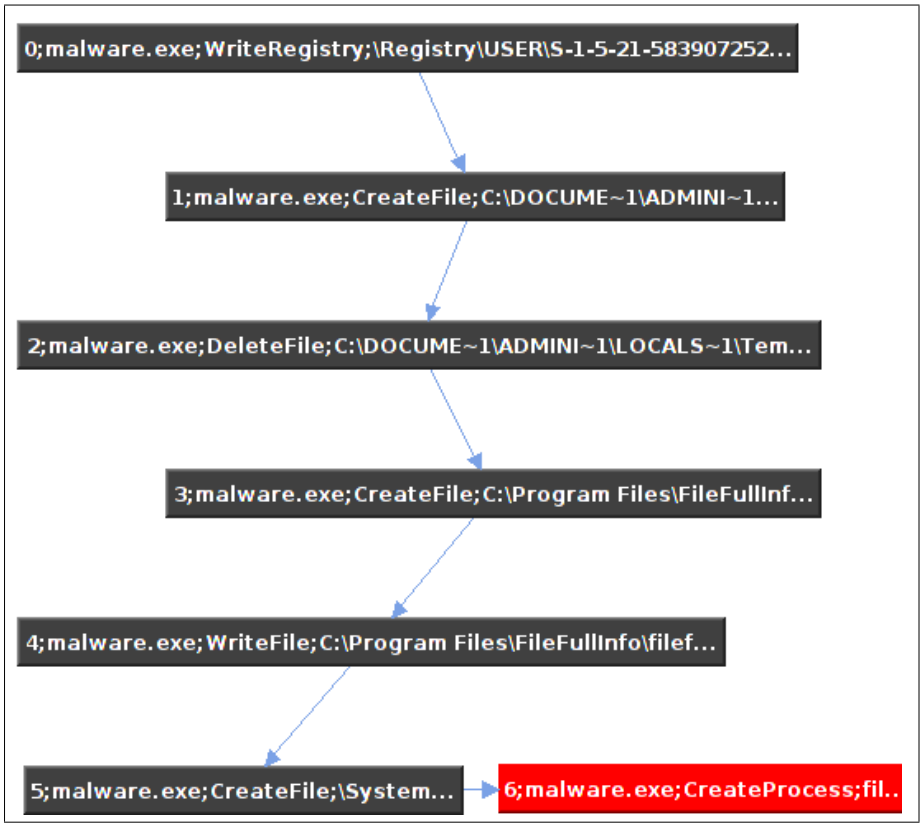

Figura 3.4: Fluxo de execução principal de um exemplar analisado, apresentado no componente "Visualizador". Durante as atividades do *malware*, ocorre a criação de um novo processo, marcado em vermelho. O processo malicioso continua sua execução normalmente.

observar quais delas foram executadas por cada processo e em qual ordem, separando assim o comportamento malicioso desenvolvido por cada um deles.

3.4 Funcionamento do sistema

Nesta seção, mostra-se como ocorre o funcionamento do sistema como um todo e apresenta-se as medidas tomadas por cada componente. Para isso, é utilizada uma situação onde um dado exemplar de *malware* é submetido para análise. São mostrados os passos realizados por BehEMOT na análise deste *malware*, através de um fluxograma, bem como um exemplo de relatório com os resultados produzidos pela análise. Na Figura 3.6 estão descritas as ações possíveis durante a análise, onde todas as alternativas estão descritas,

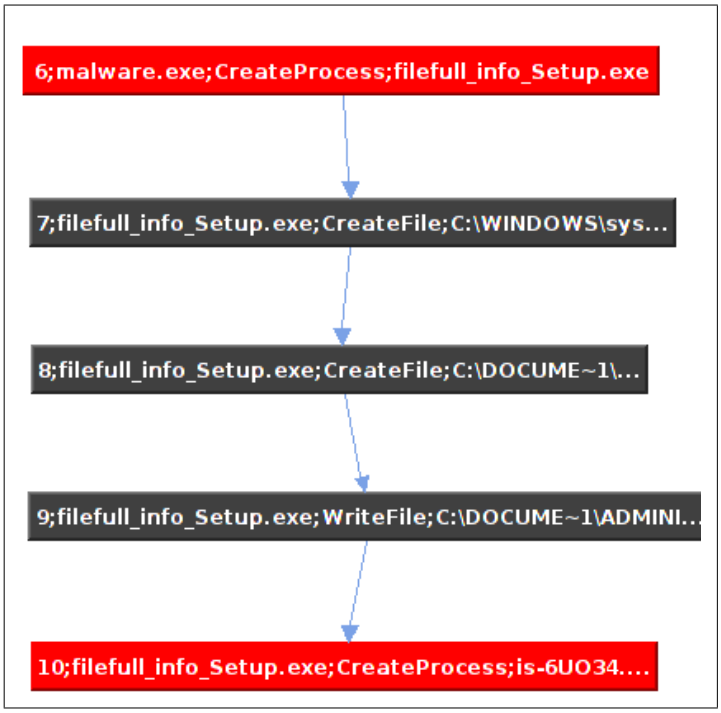

Figura 3.5: Fluxo de execução paralelo, realizado por um novo processo criado pelo exemplar de *malware* analisado mostrado na Fig. 3.4.

associadas ao componente responsável por cada uma delas.

Ao final desta análise, é gerado um relatório com as ações realizadas no sistema e o tráfego de rede capturado, já filtrados, compondo um comportamento. Um exemplo de resultado produzido pode ser observado na Figura 3.7, na qual pode-se observar as atividades geradas por meio do tráfego de rede capturado, juntamente com as atividades executadas no sistema operacional do ambiente de análise. Para facilitar a interpretação do comportamento apresentado por esse exemplar de *malware* do exemplo dado, a Figura 3.4 contém a representação produzida pelo componente visualizador.

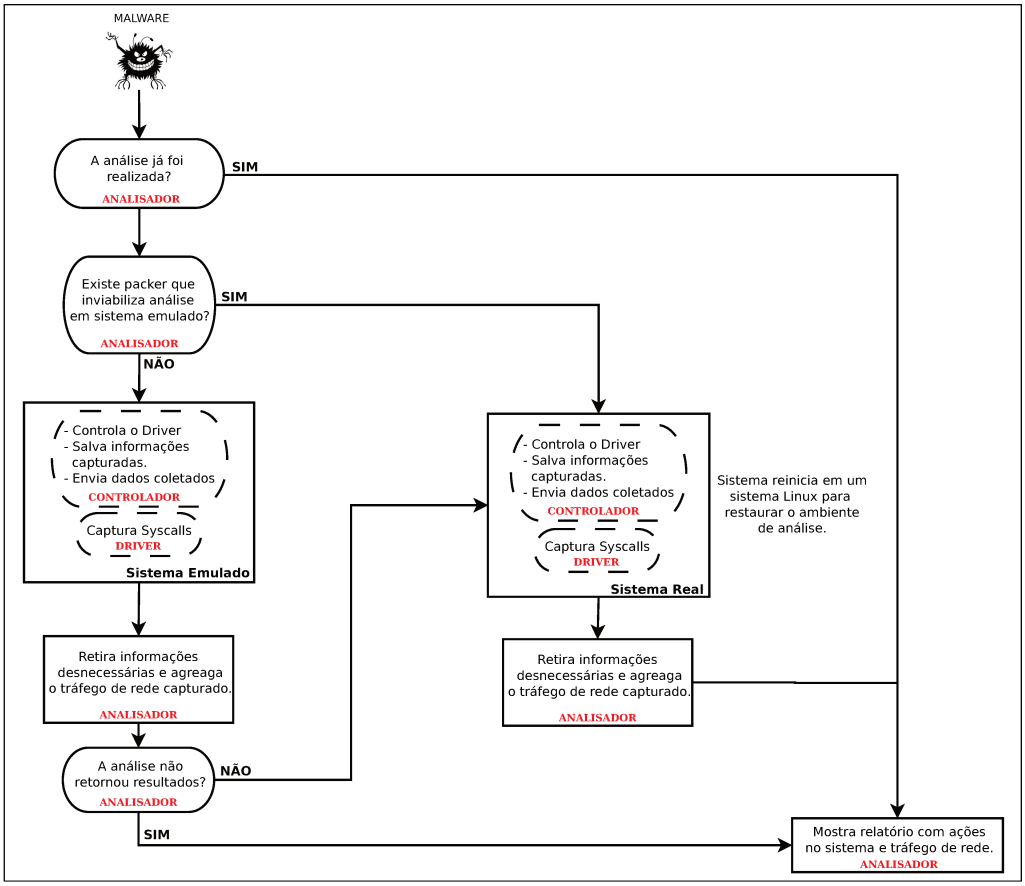

Figura 3.6: Fluxo de execução de uma análise em BehEMOT.

```
15:5:39.601;malware.exe;CreateFile;\SystemRoot\AppPatch\sysmain.sdb
15:5:39.601;malware.exe;CreateProcess;C:\Program Files\FileFullInfo\filefull_info_Setup.exe
15:5:40.31;filefull_info_Setup.exe;CreateFile;C:\WINDOWS\system32\netmsg.dll
15:5:40.512;filefull_info_Setup.exe;CreateFile;C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\is-
CUBGJ.tmp\is-6UO34.tmp
15:5:45.309;filefull_info_Setup.exe;WriteFile;C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\is-CUBGJ.tmp\is-
6UO34.tmp
15:5:46.150;filefull_info_Setup.exe;CreateProcess;C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\is-
CUBGJ.tmp\is-6UO34.tmp
15:5:47.652;is-6UO34.tmp;CreateFile;C:\WINDOWS\system32\netmsg.dll
15:5:47.853;is-6UO34.tmp;CreateFile;C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\is-
EKCIA.tmp\_isetup\_RegDLL.tmp
15:5:47.853;is-6UO34.tmp;WriteFile;C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\is-
EKCIA.tmp\_isetup\_RegDLL.tmp
15:5:47.853;is-6UO34.tmp;CreateFile;C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\is-
EKCIA.tmp\_isetup\_shfoldr.dll
15:5:47.853;is-6UO34.tmp;WriteFile;C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\is-
EKCIA.tmp\_isetup\_shfoldr.dll
15:5:47.853;s-6UO34.tmp;CreateFile;C:\WINDOWS\system32\shfolder.dll
15:5:48.63;is-6UO34.tmp;WriteRegistry;\REGISTRY\USER\S-1-5-21-583907252-2111687655-1708537768-
500\Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders\Programs
15:5:48.424;is-6UO34.tmp;CreateFile;C:\WINDOWS\system32\shell32.dll
15:5:48.424;is-6UO34.tmp;CreateFile;C:\Program Files\FileFullInfo\unins000.dat
15:5:48.484;is-6UO34.tmp;CreateFile;C:\Program Files\FileFullInfo\is-1K18M.tmp
15:5:48.494;is-6UO34.tmp;WriteFile;C:\Program Files\FileFullInfo\is-1K18M.tmp
```

Figura 3.7: Relatório do análise gerado por BehEMOT.

Capítulo 4

Testes e Resultados

Para avaliar o funcionamento da ferramenta proposta e validar seus resultados, foram realizados alguns testes, os quais foram divididos em duas partes. Na primeira, com foco na corretude da ferramenta, foram submetidos para análise arquivos cujo comportamento é conhecido. Já na segunda, foram comparados os resultados provenientes da análise de diversos exemplares de *malware* em BehEMOT e em outros sistemas de análise existentes. Foram selecionados de forma aleatória 1744 exemplares, dos quais 579 foram identificados com algum *packer* conhecido que apresenta problema na execução em ambiente emulado. Os exemplares foram retirados de uma extensa base de artefatos maliciosos disponível na Internet [52], de *honeypots* de baixa interatividade específicos para coleta de *malware* e de *spams* coletados dos *e-mails* do Laboratório de Administração e Segurança de Sistemas (LAS). Os *honeypots* utilizados são sistemas com a ferramenta *Dionaea*, um *honeypot* de baixa interatividade que possibilita a captura de determinados tipos de programas maliciosos através da emulação de serviços *Windows* vulneráveis [53]. Tais serviços ficam abertos para a Internet e têm por objetivo receber ataques e realizar o *download* de exemplares de *malware* que porventura sejam parte de um dado ataque. Sua utilização possibilita a análise de exemplares de *malware* recentes, os quais estão em circulação na Internet. Nas seções que se seguem serão apresentados como os testes foram realizados de forma mais detalhada, juntamente com os resultados alcançados.

Os sistemas de análise utilizados na comparação dos resultados com BehEMOT foram *Anubis* e *CWSandbox*. Dentre as ferramentas citadas na revisão bibliográfica, escolheu-se as acima mencionadas pois as demais, *Ether*, *Cuckoobox* e *Capture BAT* apresentam limitações na análise dos *malware* ou, como no caso específico de *Cuckoobox*, utiliza uma técnica de captura de informações idêntica a de um dos sistemas selecionados.

Exemplar (Packer)	Atividades Desenvolvidas
1 (Nenhum)	Cria um arquivo e o remove.
2 (Nenhum)	Cria um registro e o remove.
3 (Nenhum)	Cria um *mutex*.
4 (tELock!)	Faz uma conexão de rede.
5 (tELock!)	Cria um arquivo e escreve neste.
6 (tELock!)	Cria um processo.

Tabela 4.1: Exemplares desenvolvidos para o teste de análise, contendo atividades previamente conhecidas.

4.1 Corretude da ferramenta

Para este teste, foram desenvolvidos seis programas que executam ações comumente encontradas em *malware*. Visando testar o sistema de forma mais completa, três destes binários foram ofuscados com *packer* que inviabiliza a análise em sistema emulado, o que forçou a análise ser executada no ambiente real. Cada um dos programas desenvolvidos tem suas atividades descritas na Tabela 4.1, a qual será utilizada para comparação com os resultados obtidos da análise destes programas por BehEMOT e cujo resultado deve ser idêntico.

A escolha destas atividades baseou-se nos comportamentos normalmente apresentados por *malware* [42], compostas por ações simples, mas que estão habitualmente presentes neste tipo de programa. Cada um destes exemplares de teste foi codificado em linguagem C e compilado para plataforma *Windows*. Os exemplares foram submetidos para análise em BehEMOT e tiveram suas análises realizadas com sucesso, ou seja, nenhuma delas terminou de forma inesperada ou retornou um resultado vazio ao final da análise. Na Tabela 4.2 estão as informações retornadas em cada análise.

É possível observar, comparando as Tabelas 4.1 e 4.2, que as ações capturadas durante a execução de cada um dos programas desenvolvidos com atividades previamente conhecidas em BehEMOT correspondem ao comportamento esperado do exemplar, mesmo nos casos onde está presente algum tipo de *packer* que impossibilita a análise em ambiente emulado. Isto se dá devido ao uso da abordagem híbrida, a qual utiliza sistemas emulados e reais na análise, tornando possível a obtenção dos comportamentos de forma completa e sem nenhum tipo de problema.

4.2 Comparativo com outras ferramentas

Tendo em vista que as análises realizadas em BehEMOT são corretas do ponto de vista do comportamento apresentado em relação ao esperado, foram feitos testes comparativos

Exemplar (Packer)	**Resultado da análise**
1 (Nenhum)	C:\teste.exe;CreateFile;c:\teste.txt C:\teste.exe;DeleteFile;c:\teste.txt
2 (Nenhum)	C:\teste.exe;CreateRegistry;teste C:\teste.exe;DeleteRegistry;\registry\user\<HASH>\teste
3 (Nenhum)	C:\teste.exe;CreateMutex;teste
4 (tELock!)	C:\teste.exe;ConnectNet;1.2.3.4:80 C:\teste.exe;SendNet;tcp:1.2.3.4:80 C:\teste.exe;DisconnectNet;1.2.3.4:80
5 (tELock!)	C:\teste.exe;CreateFile;;\teste.txt C:\teste.exe;WriteFile;c:\teste.txt
6 (tELock!)	C:\teste.exe;CreateProcess;c:\WINDOWS\system32\NOTEPAD.EXE

Tabela 4.2: Resultado da análise dos exemplares desenvolvidos com atividades conhecidas em BehEMOT.

entre a ferramenta proposta e outras ferramentas de análise existentes. Os resultados das análises foram comparados para verificar a eficácia na obtenção do comportamento malicioso por BehEMOT. Nestes testes, foram utilizados dois sistemas de análise disponíveis publicamente e muito utilizados e referenciados no meio acadêmico, *Anubis* e *CWSandbox*. Por serem sistemas antigos, ambos disponibilizados em 2007, e projetos acadêmicos, os dois sistemas são muito utilizados nos trabalhos acadêmicos que envolvem análise dinâmica de *malware*. Verificando o número de citações no *Google Scholar* [1] que envolvem os artigos que descrevem os sistemas, é possível confirmar esta afirmação dado que existem 194 citações qu envolvem *CWSandbox* e 188 *Anubis*. Outro ponto abordado na escolha dos sistemas de análise foi a facilidade para obter os resultados das análises. Os sistemas *Ether* e *CuckooBox* precisavam ser instalados localmente para que fosse possível utilizá-los,enquanto que *Anubis* e *CWSandbox* não precisavam de nenhum recurso local. Além da disponibilidade, eles foram selecionados por utilizarem abordagens diferentes na monitoração do sistema, sendo possível comparar o resultado provido por BehEMOT em relação a outros métodos de análise. Como dito no início deste capítulo, foram descartadas outras abordagens descritas no Capítulo 2 pois existiam algumas limitações que inviabilizavam a comparação adequada ou devido à abordagem utilizar uma técnica de monitoração já presente em algum dos sistemas contemplados.

Ether, embora disponibilize seu código e seja declarado como uma ferramenta que não apresenta os problemas presentes na análise de *malware* com *packers* anti-emulador, possui a limitação de efetuar a análise de apenas um processo por vez, ou de monitorar todos os processos em execução no sistema. Como BehEMOT traça somente o processo

[1] http://scholar.google.com

do *malware* sob análise e eventuais processos filhos ou modificados por este através de escritas em memória, a fim de evitar ruído no resultado provido, optou-se por não utilizar a ferramenta *Ether* para comparação. *CWSandbox* e *Anubis* atuam de forma semelhante a BehEMOT, mostrando informações relativas somente ao processo do *malware*, processos-filhos criados por este e processos modificados pelo *malware* durante sua execução.

Já *Cuckoobox* não foi utilizado nos testes por se tratar de um sistema de análise que utiliza uma técnica similar à de *CWSandbox*, este último mais antigo e, consequentemente, melhor validado e com mais referências. Como descrito no Capítulo 2, *Cuckoobox* emprega a técnica de *Windows Detours* para capturar os dados produzidos pelo *malware*, enquanto *CWSandbox* se utiliza de *Inline Hooking* em nível de usuário, sendo que em ambos a captura dos dados ocorre no mesmo nivel de privilégio do *malware*. Outro ponto de semelhança está presente na forma como o *Inline Hooking* foi implementado em *CWSandbox*. Nela, o *hooking* é feito de maneira similar àquele feito com *Windows Detours*, através da inserção de uma instrução `JMP` no início da função interceptada.

Em relação ao *Capture BAT*, desconsiderou-se sua utilização nos testes comparativos por ele apresentar um comportamento semelhante a de *Ether*, onde as ações executadas por todo o sistema são exibidas na análise, não havendo a restrição aos processos que têm relação com o *malware*. Outro ponto que deve ser levado em consideração por um sistema de análise de *malware* é que as ações monitoradas devem ser abrangentes. No caso da ferramenta *Capture BAT*, estas estão restritas às modificações em arquivos, registros e processos, além do tráfego de rede capturado ser fornecido em formato `PCAP`. Tanto em BehEMOT, como em *Anubis* e *CWSandbox*, estão presentes de maneira interpretável as ações de rede, além das ações de modificação em memória e *mutexes*.

Os testes feitos com os 1744 exemplares de *malware* envolveram seu envio para os sistemas de análise escolhidos através do procedimento de submissão convencional, disponibilizado em [22] e [35]. Para facilitar o tratamento dos dados resultantes da análise, foram obtidos relatórios no formato *XML*. Na ocorrência de erro na submissão ou no retorno do relatório, o procedimento de submissão era repetido, evitando assim possíveis análises incompletas devido a problemas de conexão. Mesmo com a repetição do processo de submissão, houve submissões que não retornaram resultados, indicando uma possível limitação do sistema em questão. Na Tabela 4.3 é apresentada a porcentagem de relatórios retornados pelos sistemas de análise em relação ao total de exemplares de *malware* submetidos.

Nota-se, a partir da tabela, que BehEMOT foi o único que retornou resultado para todos os exemplares de *malware* enviados para análise. No caso de *Anubis*, o problema ocorreu em apenas duas submissões, nas quais o exemplar de *malware* sob análise possuía mecanismo de detecção de emuladores. Já *CWSandbox* falhou em vários exemplares, mesmo com a submissão repetida por três vezes. Quanto aos exemplares de *malware*

Sistema de Análise	Resultados retornados
BehEMOT	100%
Anubis	99,9%
CWSandbox	72,8%

Tabela 4.3: Quantidade de relatórios retornados em relação ao número de exemplares de *malware* submetidos, por sistema avaliado.

que não retornaram resultados a partir da análise por *CWSandbox*, foi possível perceber que não houve relação quanto à presença ou não de *packer* com o objetivo de evadir sistemas emulados, haja visto que grande parte dos *malware* que apresentaram problemas não contém estes tipos de *packer*. Nos testes seguintes que envolvem *CWSandbox*, o valor total de artefatos considerados para contabilizar a comparação entre as ferramentas é igual à quantidade de relatórios obtidos com sucesso, a fim de normalizar os resultados.

Para verificar a eficácia dos resultados retornados por BehEMOT perante os outros sistemas de análise, foi feita uma comparação entre os relatórios onde o menor relatório servia de base, sendo verificada a parcela de suas ações que estavam presentes no relatório maior. Entende-se por relatório menor aquele que resultou em menos ações durante a execução do *malware* no sistema de análise.

4.2.1 Normalização dos relatórios

Dado que os relatórios possuem padrões e formatos diferentes na apresentação de seus resultados, foi preciso normalizá-los, fazendo com que todos seguissem um padrão que viabilizasse a comparação entre eles. O padrão utilizado segue o mesmo formato adotado por BehEMOT, ou seja, cada atividade fica em uma linha composta pelos atributos descritos na Figura 3.2, exceto pelo *timestamp*, o qual foi retirado do padrão. Como o *timestamp* das ações não está presente nos relatórios dos outros sistemas, e também levando-se em conta que um exemplar de código malicioso pode executar suas ações em momentos diferentes, este atributo foi descartado.

Nos relatórios normalizados foram inseridas somente as ações que têm representatividade no comportamento malicioso, excluídas eventuais informações adicionais, que podem confundir a análise das ações realizadas pelo *malware* no sistema. Um exemplo disto é a lista de todos os processos que estão em execução no ambiente, presente nos relatórios de *CWSandbox*. Esta informação não exibe relação direta com o comportamento malicioso do *malware*, podendo levar a um entendimento errado do processo de comprometimento do sistema. Um outro tipo de ação, presente nos relatórios de *Anubis*, são as que tem como alvo dispositivos do sistema, como por exemplo o \Device\TCP, utilizado em ações de rede. Elas trazem dados sobre atividades normalmente realizadas no ambiente de análise,

as quais usualmente não exibem relação com a atividade maliciosa do *malware*.

Portanto, nos relatórios normalizados só estarão presentes operações realizadas em arquivos, registros, *mutex*, ações de rede, memória e criação de processos novos que forem relacionadas ao processo do exemplar de *malware* analisado. Além desta filtragem inicial, se a operação feita durante a monitoração não causa uma modificação no ambiente de análise, como por exemplo ações de leitura e abertura de *handles* para arquivos, esta não é incluída no relatório normalizado. Com esta medida são consideradas somente as ações relevantes para a definição de um possível comportamento malicioso do *malware*.

Após essa adequação onde foi adotado um padrão para os relatórios de análise e foram retiradas as informações que não expressam as alterações realizadas pelo *malware* no sistema de análise, foram comparados os relatórios, dois a dois, para extrair a similaridade entre eles. Inicialmente utilizou-se o índice de *Jaccard* para obter este valor. O índice de *Jaccard*, quando aplicado para medir a similaridade entre dois relatórios R_{M_1} e R_{M_2}, irá expressar quantas ações semelhantes existem entre eles, sendo que uma ação semelhante é composta por um executor, tipo de ação e alvo idênticos. A fórmula que descreve a similaridade de *Jaccard* é:

$$J(R_{M_1}, R_{M_2}) = \frac{R_{M_1} \cap R_{M_2}}{R_{M_1} \cup R_{M_2}}, 0 \leq J \leq 1$$

Nas comparações, os conjuntos utilizados são representados pelos relatórios ajustados de cada ferramenta para um determinado *malware*, composto pelas ações descritas durante sua execução no ambiente de análise. Os resultados obtidos desta comparação inicial mostravam valores muito baixos, chamando atenção para a alta concentração de relatórios com similaridade entre 0 a 10%. A Tabela 4.4 mostra estes valores separados entre intervalos de 10% para a similaridade presente entre relatórios de *malware* que não contêm *packer* anti-sistema emulado. Analisando manualmente alguns relatórios que estavam na faixa entre 0 a 10% verificou-se que muitos deles apresentavam muitas ações semelhantes, diferindo apenas alguns pequenos detalhes, como por exemplo, o nome do processo executor da ação ou alguma chave de registro que fazia referência à variáveis do ambiente, as quais eram diferentes para cada sistema de análise. Outro ponto que estava influenciando na alta concentração das similaridades na faixa mais baixa era devido aos relatórios apresentarem tamanhos diferentes. Para obter o índice de *Jaccard* que irá representar a similaridade entre dois relatórios, é necessário dividir o valor que corresponde ao número de elementos presentes na intersecção dos dois relatórios pelo valor que corresponde ao tamanho do conjunto resultante na união deles. Caso os relatórios tenham tamanhos muito diferentes, com um relatório com 20 ações e o outro com 2 por exemplo, o índice de *Jaccard* resultará em um valor muito pequeno, mesmo que o relatório menor esteja totalmente contido no relatório maior. Neste ponto, percebeu-se que a melhor medida para se calcular a similaridade entre dois relatórios não poderia ser o índice de

Faixas	Anubis × BehEMOT	CWSandbox × BehEMOT	Anubis × CWSandbox
0-10%	309	633	583
10-20%	262	53	45
20-30%	138	17	15
30-40%	115	4	4
40-50%	87	4	3
50-60%	26	3	5
60-70%	9	3	1
70-80%	4	0	0
80-90%	0	2	0
90-100%	12	7	3

Tabela 4.4: Faixas de porcentagem que representam a similaridade encontrada entre os relatórios através do índice de *Jaccard* para a análise de *malware* que não apresentam *packers* que levem a problemas em ambiente emulado. Os valores mostrados representam a quantidade de relatórios que têm similaridade dentro de uma determinada faixa.

Jaccard e sim a porcentagem que representa o quanto do relatório menor está contido no maior. Usando esta métrica, que representa a taxa de inclusão do relatório menor no maior, mesmo que um sistema capture menos ações durante a análise, o valor da taxa de inclusão entre eles não será afetada pela diferença no tamanho dos relatórios.

Mesmo utilizando esta nova métrica foi preciso realizar algumas modificações nos valores de alguns atributos, para que as comparação entre os relatórios ocorressem de forma correta. Ações que tinham como alvo arquivos modificados/criados durante a análise, que normalmente variam de acordo com a execução do *malware*, tiveram seus atributos ajustados para valores do tipo *file.*<posição do arquivo criado na tabela *hash* de arquivos criados>. Sempre que um novo arquivo aparecia em um alvo, uma tabela *hash* com todos os nomes de arquivos encontrados até o momento era consultada. Se o arquivo não estava presente ele era adicionado à tabela e o valor do alvo era modificado para *file.*<número de arquivos na tabela *hash* + 1>. Caso contrário, o alvo era modificado para *file.*<posição do arquivo criado na tabela *hash* de arquivos criados>. O mesmo princípio de nomeação utilizado nos nomes dos arquivos foi aplicado também aos nomes dos processos presentes nos relatórios, quer seja no campo executor da ação ou alvo.

Além dos problemas com os nomes aleatórios, alguns alvos presentes nas ações envolvendo arquivos fazem referência a *handles* do sistema operacional, utilizados em operações normais como por exemplo, tráfego de rede gerado pela máquina. Os mais comuns e conhecidos foram retirados do relatório normalizado, de modo que estas ações não gerem ruídos considerados atividades normais do sistema.

Outro tipo de ação que teve o campo "alvo" filtrado foi as que envolvem registros.

Faixas	Anubis × BehEMOT	CWSandbox × BehEMOT	Anubis × CWSandbox
0-10%	112	239	210
10-20%	70	73	56
20-30%	81	117	67
30-40%	135	74	54
40-50%	85	39	41
50-60%	110	88	109
60-70%	131	29	49
70-80%	78	18	18
80-90%	47	5	14
90-100%	115	44	41

Tabela 4.5: Faixas de porcentagem que representam a taxa de inclusão encontrada nos relatórios normalizados entre os sistemas para a análise de *malware* que não apresentam *packers* que levem a problemas em ambiente emulado. Os valores mostrados representam a quantidade de relatórios com taxa de inclusão calculada pela métrica definida anteriormente.

Certos registros do sistema são vinculados ao usuário que está usando a máquina, sendo diferentes para cada ambiente de análise. Caso o *malware* realize alguma ação em um registro que esteja atrelado ao usuário do ambiente, a comparação não terá sucesso, dado que os valores são diferentes em cada ambiente. Visando eliminar este problema durante a normalização, os valores foram modificados para um identificador estático <RID>.

Atividades de rede também apresentavam problemas quando o *malware* fazia referência a diferentes endereços IP. Este tipo de ação fica evidente quando um *malware* realiza um *scan*, onde são verificados um grande número de endereços IP pertencentes a uma determinada rede durante a execução do *malware*. Como nos casos anteriores, uma simples modificação do endereço IP para um valor estático, <IP>, resolveu o problema. O valor da porta destino da conexão foi mantido, para diferenciar as tentativas de conexão a serviços diferentes.

Após realizar todas estas modificações, as linhas duplicadas presentes no relatório resultante eram removidas para evitar comparações incorretas. Na Tabela 4.5 são apresentados os resultados obtidos aplicando-se a nova métrica de cálculo da taxa de inclusão nos relatórios normalizados. Entende-se por relatórios normalizados aqueles onde são aplicadas as alterações explicadas anteriormente.

É possível perceber que os resultados obtidos melhoraram consideravelmente. Um fator que contribuiu para esta modificação foi a métrica usada para calcular a taxa de inclusão. Agora, as diferenças existentes entre os tamanhos dos relatórios não influencia na taxa de inclusão, como anteriormente. Já as outras modificações, que visavam deixar o nome de processos e arquivos mais genérico, não tiveram um efeito tão positivo. Ve-

rificando manualmente alguns relatórios normalizados com taxa de inclusão na faixa de 0 a 10%, percebeu-se que muitos relatórios apresentavam muitas ações semelhantes, diferindo apenas pelo nome do processo que executou a ação. Alguns *malware* executavam certas ações no processo inicial do *malware* em um relatório enquanto que nos outros, as mesmas ações eram executadas por um outro processo, iniciado pelo *malware*. No momento da comparação, várias ações iriam ser identificadas como diferentes, mesmo que fossem iguais. A ordem em que apareciam os processos no relatório era importante para determinar o novo nome do processo, que estaria presente no relatório normalizado. Por isso, mesmo com essa generalização, não foi possível tornar os nomes genéricos o bastante para evitar comparações erradas. Este mesmo efeito foi percebido no nome dos arquivos criados pelo *malware*, resultando assim em mais comparações incorretas.

Uma forma de contornar estes problemas foi utilizar identificadores estáticos, tanto para os nomes de processo quanto para arquivos. Portanto, sempre que um nome de processo aparecia, ele era trocado para o identificador *process* e os de arquivos para *file*. Para evitar que após a normalização os relatórios resultantes omitissem ações realizadas pelo *malware*, dado que as linhas repetidas são removidas, separou-se o processo de normalização em duas etapas. Na primeira, os nomes de registro que apresentavam problemas e as conexões de rede eram modificados de acordo com as definições explicadas anteriormente. Ainda na primeira etapa, todas as ações duplicadas eram removidas do relatório gerado após as modificações, retirando assim casos onde o *malware* realizava várias conexões de rede à um mesmo serviço, por exemplo em um *scan*. A etapa seguinte modificava os nomes de processos e arquivos para valores estáticos sem a remoção de ações duplicadas. A não remoção destas ações na segunda etapa evitou que a criação de vários arquivos fosse resumida a uma só ação, dado que no relatório normalizado essas várias escritas aparecem de forma idêntica.

Os resultados obtidos foram bem melhores que os anteriores e expressaram de forma correta a taxa de inclusão presente entre os relatórios normalizados comparados. Todos os valores alcançados com estas modificações serão apresentados na subseção seguinte. A seguir serão explicitadas todas as modificações realizadas nos relatórios originais. As ações presentes nos relatórios normalizados estão expostas na Tabela 4.6, bem como a ocorrência de filtragens para efeitos de normalização. Na Tabela 4.7 são apresentadas todas as filtragens efetuadas nos relatórios, detalhando-se como as ações eram representadas antes da filtragem e como ficaram após sua aplicação.

Após gerar todos os novos relatórios, foram descartados aqueles que estavam vazios, dado que não seria possível compará-los. A Tabela 4.8 apresenta a porcentagem de relatórios normalizados vazios em relação à quantidade de relatórios originais, separando-os de acordo com a presença ou não de *packers* conhecidos com mecanismo anti-emuladores. Pode-se perceber que a presença do *packer* não está relacionada diretamente com o su-

Ações	**Anubis**	**BehEMOT**	**CWSandbox**
Leituras de Registro/Arquivos	R	R	R
Modificações em Registro/Arquivos	F	F	F
Escritas de Memória	F	F	F
Operações de Rede	P	P	P
Operações com *Mutex*	P	P	P
Criação de novos processos	F	F	F
Informações adicionais	R	R	R

Tabela 4.6: Ações presentes nos relatórios de análise normalizados. Os valores possíveis são: F, indicando que a ação foi filtrada do relatório normalizado; P, indicando que a ação existia no relatório original e foi mantida; e R, que denota quando uma ação presente no relatório original foi omitida do normalizado.

cesso ou não na obtenção das ações maliciosas. Um possível motivo disto é a limitação quanto à detecção da presença de *packer*. Como ela é feita baseada em assinaturas de *packers* conhecidos, é possível que a versão utilizada da base de assinaturas não contenha alguns *packers* não tão comuns ou não possua uma assinatura para outros já conhecidos.

Das análises geradas por *Anubis* sobre exemplares de *malware* que contêm *packer* com mecanismo anti-emulador, 297 deles retornaram relatórios vazios após serem normalizadas. Ao ser feita a verificação destes mesmos exemplares em BehEMOT observou-se que todos eles efetuavam algum tipo de ação, tais como criação de arquivos e tráfego de rede por exemplo. Considerando os 297 exemplares executados em *Anubis* que retornaram relatórios normalizados vazios, pôde-se verificar que cerca de 112 deles fizeram algum tipo de ação quando executados em *CWSandbox*. Já o contrário, ou seja, exemplares de *malware* que retornaram relatórios normalizados vazios em *CWSandbox*, dos 60 que não tinham qualquer atividade, 43 realizaram algum tipo de atividade quando executados em *Anubis*. Vale ressaltar também que *CWSandbox* retornou menos resultados válidos do que *Anubis*.

Nas análises realizadas em BehEMOT, verificou-se que 6 relatórios normalizados estavam vazios. Comparando-os com os de *Anubis*, verificou-se que todos eles realizaram atividades em *Anubis*, porém elas se restringiam apenas à criação de *mutexes*. Na comparação com *CWSandbox*, 5 dos relatórios normalizados vazios de BehEMOT realizaram algum tipo de ação em *CWSandbox*. Como em *Anubis*, os resultados obtidos de *CWSandbox* que apresentaram problemas em BehEMOT mostravam somente ações de criação de *mutexes*.

Separados os relatórios vazios, iniciou-se o processo de comparação entre os relatórios normalizados produzidos por cada um dos sistemas. Cada relatório era comparado automaticamente, obtendo ao fim a porcentagem referente à quantidade de ações do menor

Filtragem	Antes	Depois
Nomes dos arquivos criados ou modificados	malware.exe;CreateFile;c:\a.txt	malware.exe;CreateFile;file
Nomes dos processos criados ou modificados	malware.exe;CreateProcess;c:\teste.exe	malware.exe;CreateProcess;process
Nomes dos processos modificados para valor estático	malware.exe;CreateProcess;c:\teste.exe	process;CreateProcess;process
Nomes de arquivos criados ou modificados que fazem referência a *handles* do sistema operacional	malware.exe;CreateFile;\Device\Ip	-
Registros criados ou modificados que fazem referência a variáveis do ambiente	malware.exe;WriteRegistry;{8cdfc5cf-4646-d9fc-7a57-cf1f626f4e7a}	malware.exe;WriteRegistry;<HASH>
Conexões de rede	malware.exe;ConnectNet;10.0.0.1:80	malware.exe;ConnectNet;<IP>:80

Tabela 4.7: Relação de todas as filtragens realizadas nos relatórios para que a comparação fosse realizada de forma normalizada. As colunas mostram como era antes da filtragem e como ficou a informação após.

Sistema	*Malware* com packer	*Malware* sem packer
BehEMOT	1,0%	0%
Anubis	16,2%	17,4 %
CWSandbox	3,0%	3,9%

Tabela 4.8: Porcentagens dos relatórios normalizados vazios, separados pela presença ou não de *packer* que inviabiliza a análise em ambiente emulado.

Faixas	Anubis × BehEMOT	CWSandbox × BehEMOT	Anubis × CWSandbox
0-10%	17	123	152
10-20%	5	32	19
20-30%	13	38	32
30-40%	17	42	33
40-50%	32	28	36
50-60%	119	101	99
60-70%	154	87	78
70-80%	186	62	44
80-90%	118	73	52
90-100%	303	140	122

Tabela 4.9: Faixas de porcentagem que representam a taxa de inclusão encontrada nos relatórios normalizados entre os sistemas para a análise de *malware* que não apresentam *packers* que levem a problemas em ambiente emulado. Os valores mostrados representam a quantidade de relatórios que têm taxa de inclusão entre as faixas descritas.

relatório contidas no relatório maior. O menor relatório era aquele com menos ações capturadas durante a execução do *malware*, e similarmente para o maior. Utilizou-se como referência o maior relatório devido a ele representar o sistema que aparentemente capturou mais informações durante a análise. Um possível motivo para o outro relatório ser menor seria algum problema existente no sistema que tenha impossibilitado a geração do comportamento completo, seja ele devido a alguma limitação da ferramenta ou ao *malware* apresentar um comportamento diferenciado.

Em cada análise, os conjuntos utilizados nas comparações são representados pelos relatórios normalizados de cada ferramenta para um determinado *malware*, composto pelas ações descritas por ele durante sua execução no ambiente de análise. Portanto, o valor encontrado irá expressar quantas ações semelhantes do relatório menor existem no maior, sendo que uma ação semelhante é composta por um executor, tipo de ação e alvo idênticos.

Os testes envolvendo as taxas de inclusão dos resultados foram divididos entre os *malware* que possuíam algum *packer* anti-emulador e aqueles que não foram identificados com este tipo de *packer*. Os resultados comentados a seguir são referentes aos exemplares que foram identificados como ausentes de qualquer tipo de *packer* que apresente dificuldade na execução em ambiente emulado.

4.2.2 *Malware* sem *packer* anti-sistema emulado

A Tabela 4.9 e os gráficos nas Figuras 4.1 e 4.2 mostram as taxas de inclusão entre os relatórios normalizados dos sistemas de análise para cada *malware* analisado. Nestes

exemplares, estão incluídos *malware* que não contém nenhum dos *packer*, descritos na literatura, que apresentam problemas na execução em ambiente emulado (*tElock!*, *Armadillo* e *ASProtect*. Alguns exemplares continham *packers* porém de outros tipos. Vale ressaltar que a ferramenta utilizada para identificar os *packers* se baseia em assinaturas, ou seja, há possibilidade de algum *malware* contêr um *packer* que ainda não foi identificado, o qual pode inviabiliar a analise em ambiente emulado. Na Tabela 4.9, as taxas de inclusão foram separadas entre faixas, de forma a apresentar os valores absolutos obtidos nas comparações. O gráfico apresentado na Figura 4.1 apresenta os dados obtidos na Tabela 4.9. Como na Tabela 4.9, os relatórios que continham taxa de inclusão idêntica foram agrupados e a frequência de ocorrência deles acumulada, mas agora dividida pelo número total de relatórios normalizados existentes. Mesmo assim, ele não consegue mostrar os resultados de uma forma clara, que demonstrem quais sistemas têm relatórios mais semelhantes entre si. O gráfico da Figura 4.2 visa clarificar isto, apresentando os resultados de uma forma mais explícita. No eixo Y é apresentada a frequência de ocorrência de uma determinada taxa de inclusão, de maneira cumulativa. Para isto, o valor da frequência da taxa de inclusão começa a ser plotado a partir de 100%, decrescendo a 0%. Em tal gráfico, sistemas que tenham alta taxa de inclusão nos seus relatórios terão curvas que se aproximam rapidamente de 100% no eixo Y. Ilustrando a interpretação das curvas neste gráfico, o primeiro ponto plotado, i.e. aquele que encosta no eixo Y, indica a porcentagem dos relatórios que têm taxa de inclusão igual a 100%. Conforme o parâmetro no eixo X vai diminuindo, i.e. à medida que se vai considerando relatórios progressivamente com taxas de inclusão diferentes , o valor no eixo Y tenderá a chegar a 100%, indicando a consideração da totalidade dos casos.

Observando a Figura 4.2, é possível perceber que BehEMOT têm mais relatórios normalizados em comum com *Anubis* do que com *CWSandbox*, dado que grande parte dos relatórios normalizados tem taxas de inclusão mais altas. Quando comparados com os resultados de *CWSandbox*, ambos sistemas *Anubis* e BehEMOT têm grande parte dos relatórios com baixa taxa de inclusão. No gráfico, é possível ver que o número de exemplares com taxa de inclusão 100% é menor em ambos os casos. Além disto, conforme o número de relatórios considerados vai aumentando, a taxa de inclusão acumulada entre BehEMOT e *Anubis* vai se distanciando das curvas entre cada um deles e *CWSandbox*. Na Tabela 4.9 existe um grande número de exemplares presente na faixa de 0 a 10% nas comparações que envolvem *CWSandbox*.

Verificando individualmente os relatórios presentes dentro desta faixa, percebeu-se que, nos casos onde o *malware* apresentou alguma falha durante a execução, a qual era tratada pela aplicação *DrWatson*, *CWSandbox* apresentou uma única ação. Já em BehEMOT e *Anubis* foi possível verificar as ações executadas pelo *malware* até o erro acontecer e também as atividades feitas pelo *DrWatson*. Estas ações não podem ser descartadas, pois

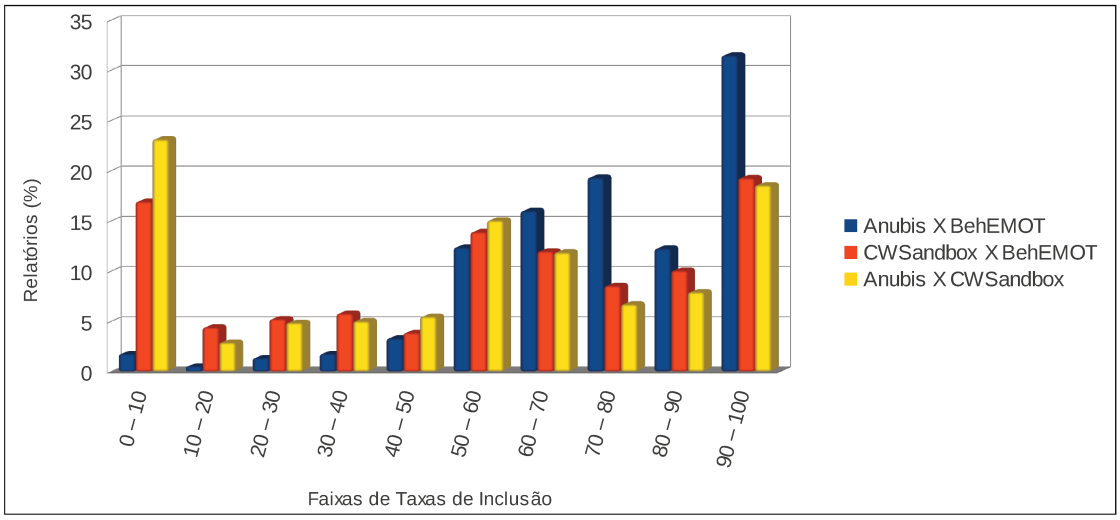

Figura 4.1: Faixas de porcentagem das taxas de inclusão entre os resultados obtidos da análise dos exemplares de *malware* que não contêm *packer* conhecido com características de anti-emulador. Os valores mostrados representam a porcentagem de relatórios com taxas de inclusão que se enquadram entre as faixas descritas, c.f. Tabela 4.9.

o *malware* pode modificar o binário do *DrWatson* para que ele se comporte de maneira maliciosa ao ser executado. Neste caso, tanto *Anubis* quanto BehEMOT iriam capturar as ações provenientes da modificação do *malware*, enquanto que *CWSandbox* não.

Verificou-se também, manualmente, os relatórios que obtiveram taxas de inclusão na faixa de 0 a 10% entre *Anubis* e BehEMOT. Percebeu-se que, na maioria das vezes, as ações executadas pelo *malware* em *Anubis* e em BehEMOT eram bem diferentes. Em um relatório de *Anubis* notou-se uma atividade que pode caracterizar alguma falha no momento da geração do relatório por parte de *Anubis*. Uma chave de registro criada tinha como alvo um valor incompleto, apresentando somente caracteres "\", enquanto que em BehEMOT o resultado mostrava um valor com registro completo. Isto demonstra que as baixas taxas de inclusão provenientes de comparações entre os relatórios de *Anubis* e BehEMOT são resultado de comportamentos diferentes, apresentados entre as execuções dos *malware* nos sistemas considerados. Diferente disto, muitas das comparações que envolvem *CWSandbox* e têm baixas taxas de inclusão, ocorrem devido à captura incompleta das ações do *malware*, dado que toda vez que o *DrWatson* é invocado, *CWSandbox* não obtêm todos os dados produzidos pelo sistema.

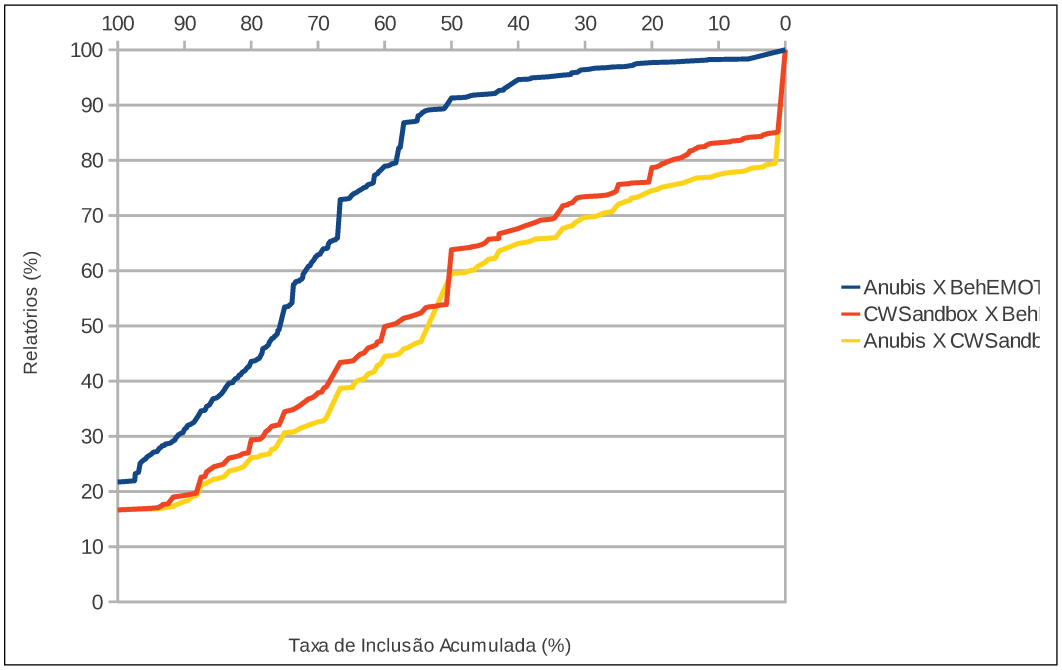

Figura 4.2: Comparativo entre as taxas de inclusão dos resultados obtidos da análise dos exemplares de *malware* que não contêm *packer* conhecido com características de anti-emulador.

4.2.3 *Malware* com *packer* anti-sistema emulado

Na segunda etapa dos testes, foram analisados somente exemplares de *malware* que continham algum *packer* reconhecido pelo *PEiD*[2] [50] que tornava a análise inviável em sistema emulado. Neste conjunto de 579 exemplares de *malware*, havia *packers* do tipo *tElock!, Armadillo e ASProtect*. A Tabela 4.10 mostra as faixas de taxas de inclusão entre os relatórios normalizados dos sistemas e a Figura 4.3 explicita graficamente os dados da tabela. O gráfico da Figura 4.4 segue os mesmos parâmetros do apresentado na Figura 4.2, sendo modificados apenas os valores referentes aos pontos do gráfico. Pode-se perceber que a distribuição das taxas de inclusão ocorre de forma análoga à dos *malware* sem *packer* problemático, sendo os relatórios de *Anubis* e BehEMOT mais semelhantes entre si do que entre cada um deles e *CWSandbox*. Um fato interessante é a grande presença de relatórios com taxas de inclusão na faixa mais alta, de 90 a 100%. Verificou-se que uma parcela significativa dos relatórios entre *Anubis* e BehEMOT que estavam nesta faixa eram de *malware* com *packer tElock!*. Na comparação entre BehEMOT e *CWSandbox*, a distri-

[2]Portable Executable Identifier

Faixas	Anubis × BehEMOT	CWSandbox × BehEMOT	Anubis × CWSandbox
0-10%	4	24	27
10-20%	8	9	16
20-30%	2	32	23
30-40%	9	42	27
40-50%	2	12	13
50-60%	23	59	49
60-70%	30	57	27
70-80%	53	31	18
80-90%	56	76	42
90-100%	296	139	192

Tabela 4.10: Faixas de porcentagem que representam a taxa de inclusão encontrada nos relatórios entre os sistemas para a análise de *malware* que contêm *packer* anti-emulador. Os valores mostrados apresentam a quantidade de relatórios que têm taxa de inclusão dentro das faixas descritas.

buição entre as faixas de taxas de inclusão ficou mais uniforme, sendo que os relatórios se concentraram na faixa de taxas de inclusão mais alta, de 90 a 100%; o mesmo ocorreu entre *Anubis* e *CWSandbox*, embora com menor taxa de inclusão acumulada. Muitos relatórios dentro desta faixa também eram relativos a *malware* com *tElock!*.

Verificando manualmente os exemplares que continham *tElock!* e estavam na faixa de 90 a 100%, constatou-se que estes *malware* não foram analisados de forma correta por *Anubis*, tendo suas ações capturadas somente por BehEMOT e *CWSandbox*. Isto porque estes relatórios apresentavam uma ação de criação de *mutex* com nome de 8 caracteres, sendo que poucos deles continham mais ações, as quais também se tratavam de criações de *mutexes*. Somente em um relatório percebeu-se mais ações, as quais também estavam presentes nos relatórios de BehEMOT e *CWSandbox*. Analisando os relatórios desta mesma faixa quando comparados BehEMOT e *CWSandbox*, verificou-se que em alguns casos *CWSandbox* não capturou algumas ações de criações de registros que aparecem em BehEMOT. Um dos possíveis motivos para este comportamento desigual é alguma limitação presente em *CWSandbox*, que impossibilitou a captura do comportamento mais completo.

No gráfico apresentado na Figura 4.5 é possível verificar como ficou a distribuição entre as faixas de taxas de inclusão quando compara-se somente *malware* com *tElock!*. Fica evidente que quase todos os relatórios de *Anubis* têm taxa de inclusão dentro da faixa de 90 a 100%, tanto para comparações com BehEMOT quanto com *CWSandbox*. Já nas comparações entre BehEMOT e *CWSandbox* os valores não seguem esta regra, dado que *CWSandbox* consegue capturar mais ações provenientes de uma das possíveis execuções do *malware*.

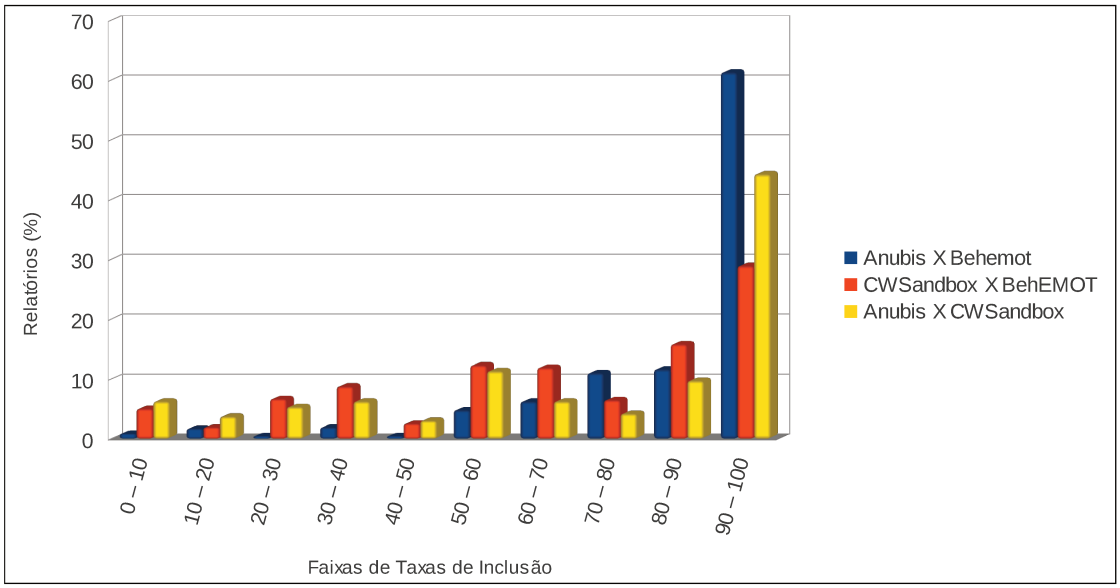

Figura 4.3: Faixas de porcentagem das taxas de inclusão entre os resultados obtidos da análise dos exemplares de *malware* que contêm *packer* conhecido com características de anti-emulador. Os valores mostrados representam a porcentagem de relatórios com taxas de inclusão que se enquadram entre as faixas descritas, c.f. Tabela 4.10.

4.3 Desempenho

Apesar de BehEMOT apresentar bons resultados quando comparado com outras ferramentas relevantes de análise de *malware*, a sua performance pode ser pior em determinados casos. Caso o *malware* não apresente qualquer comportamento maligno durante a sua execução em ambiente emulado, ele será executado novamente no ambiente real, o que fará com que o tempo de análise seja no mínimo 2 vezes maior, se comparado com as demais ferramentas. Mesmo com esta limitação, o tempo de espera total ainda é aceitável para uma ferramenta de análise dinâmica como a proposta. Outro problema existente é a necessidade de máquinas reais. Se compararmos com as abordagens que utilizam emuladores ou máquinas virtuais, nas quais é possível gerar várias análises simultâneas a partir de uma única máquina, BehEMOT pode não ser tão flexível, pois cada máquina real analisa apenas um exemplar de *malware* por vez. Entretanto, o uso de máquinas reais torna possível a análise de exemplares com mecanismos de detecção de máquinas virtuais e emuladores, o que não é factível nos sistemas citados. Além disso, com o uso da abordagem híbrida, que mescla análises em sistemas emulados e reais, esta limitação é

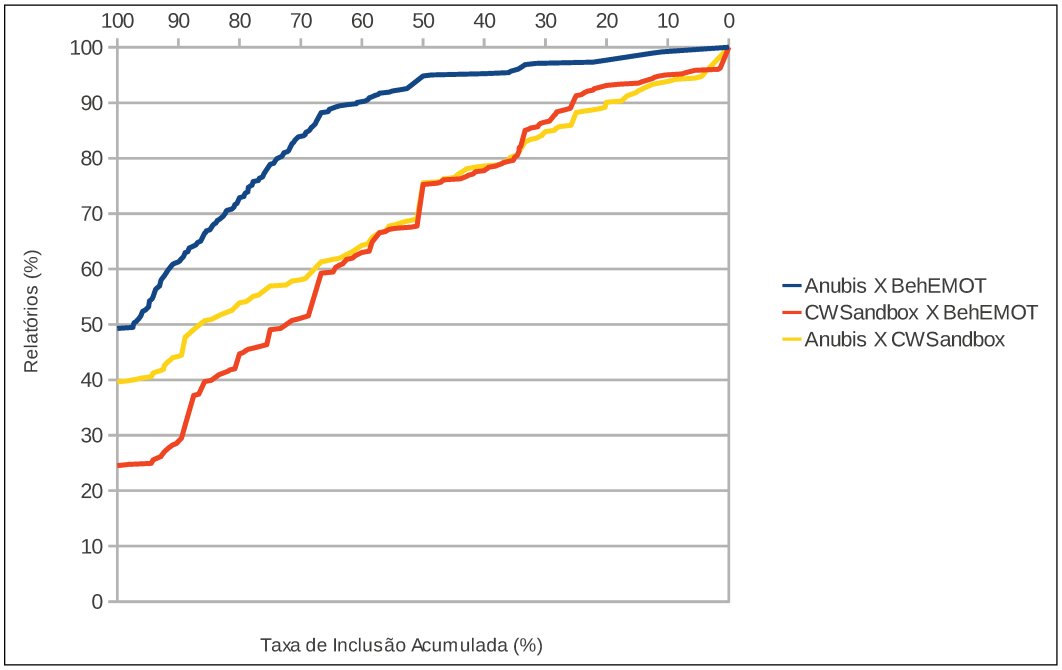

Figura 4.4: Comparativo entre as taxas de inclusão dos resultados obtidos da análise dos exemplares de *malware* que contêm *packer* conhecido com características de anti-emulador.

amenizada, dado que não é necessária uma máquina real para cada ambiente de análise. Outro ponto favorável é que, com o uso mais frequente do ambiente emulado, o *overhead* da cópia da imagem de disco do sistema real se torna menos significativo, visto que tal operação só será necessária nos casos de *malware* mais sofisticados.

4.4 Detecção da Ferramenta

Mesmo realizando a captura em um nível de privilégio superior ao de execução do *malware*, é possível para um exemplar de *malware* detectar BehEMOT. Como existe um outro componente do sistema—o "controlador"—que executa no mesmo nível de privilégio do *malware*, basta uma simples busca dentre os processos do sistema para identificar BehEMOT. Entretanto, o "*driver*" no nível do *kernel* pode ser estendido para utilizar técnicas que visem esconder componentes de nível de usuário, tornando-os imperceptíveis a programas no mesmo nível. Porém, se o *malware* usar um programa com privilégio de *kernel*, mesmo os componentes ocultos podem ser identificados.

Para mostrar que mesmo sistemas que realizam a captura das ações do *malware* fora

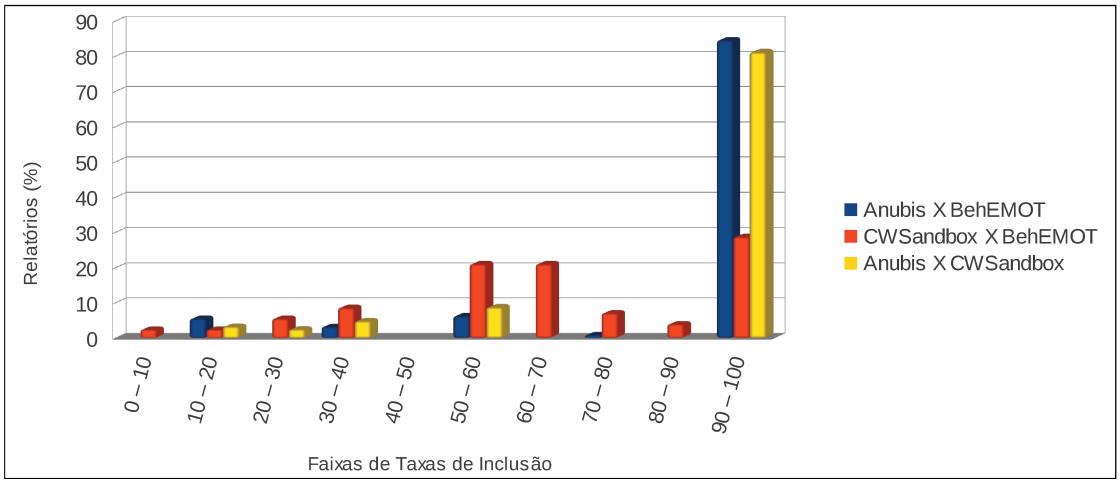

Figura 4.5: Comparativo entre as taxas de inclusão dos resultados obtidos da análise dos exemplares de *malware* que contêm *packers* identificados com anti-emulador *tElock!*.

do ambiente do sistema de análise deixam evidências, foram realizados alguns testes em *Anubis* para verificar possíveis rastros de monitoração. Os testes foram feitos com o objetivo de procurar por programas e documentos existentes na raiz do sistema operacional que não fazem parte da instalação padrão. Caso fosse encontrado algum arquivo diferente, este seria enviado via rede para uma máquina local, para posterior análise. Este simples teste identificou arquivos executáveis e de inicialização utilizados por *Anubis*, os quais não estão aqui apresentados por caracterizarem informações confidenciais do sistema, fugindo assim do escopo desta dissertação.

Conclui-se portanto que, mesmo com o privilégio adicional da utilização da técnica de VMI[3] existente no sistema *Anubis*, situado entre o sistema de análise e o componente de captura, existem dados simples (como arquivos específicos na raiz do sistema de análise) que podem identificar facilmente o ambiente de monitoração. De forma análoga, é possível utilizar técnicas que detectem os componentes de BehEMOT com programas de nível de usuário, possibilitando ao *malware* mudar seu comportamento, resultando em uma análise incompleta. Entretanto, como explicado anteriormente, é possível estender BehEMOT de forma a tornar mais difícil a detecção do ambiente de análise. Além disso, a utilização de máquinas reais permite a análise de exemplares de *malware* que evadem a monitoração feita em ambientes emulados ou virtualizados, como é o caso dos demais sistemas avaliados.

[3] *Virtual Machine Introspection*

Capítulo 5

Conclusão

A análise de *malware* desempenha um papel muito importante no processo de identificação de novas ameaças, sendo uma ferramenta fundamental no auxílio ao desenvolvimento de métodos de combate a exemplares de *malware* mais recentes. O uso de assinaturas tem se mostrado uma prática com muitos problemas, principalmente relacionados ao crescente número de variantes de *malware* que surgem diariamente. Este aumento demanda atualizações frequentes ao banco de assinaturas dos mecanismos de antivírus, além de exigir que os novos exemplares obtidos sejam analisados mais rapidamente, de modo a gerar as proteções adequadas em tempo hábil.

Deste modo, os sistemas de análise dinâmica de *malware* surgem como um auxílio na identificação das ações que um dado *malware* realiza no sistema da vítima, facilitando a monitoração das atividades efetuadas durante sua execução em um ambiente similar ao de um usuário. Apesar de existirem sistemas disponíveis publicamente que podem ser utilizados de forma gratuita, estes apresentam algumas limitações na análise de tipos específicos de *malware*, como por exemplo a presença de algum *packer* que identifique e evada ambientes emulados ou virtuais. O diferencial da ferramenta proposta neste trabalho, BehEMOT, está em sua abordagem híbrida, que executa o *malware* em um ambiente emulado e, caso exista algum problema durante a execução, direciona-o para um ambiente real.

A escolha do sistema operacional utilizado nos ambietes de análises, *Windows XP* com *Service Pack* 3, deve-se a vários fatores sendo o primeiro deles a disponbilidade de tal sistema no momento do desenvolvimento do componente de captura. Somando-se a isto existe o fato dos sistemas de análise utilizados na comparação utilizarem *Windows XP* também **??**, **??**, porém com *Service Packs* diferentes. Felizmente, caso seja necessário, é possível modificar o componente de captura para que seja possível executá-lo em um sistema operacional mais recente, como por exemplo o *Windows* 7. Apesar de serem necessárias mudanças no código do componente de captura, não haveria muita dificuldade

em portá-lo para uma nova versão do *Windows*. O maior empecilho seria a necessidade de desabilitar o *Patch Guard*, um mecanismo de proteção implementado nas versões recentes do *Windows*, que detecta modificações feitas na tabela *SSDT* [1] e reinicia o sistema como medida de proteção. Mesmo assim, é possível desabilitar esta proteção através de técnicas já disponibilizadas, e assim modificar a *SSDT* sem problemas.

Nos resultados do teste de corretude, foi possível verificar o funcionamento adequado da ferramenta, dado que BehEMOT pôde monitorar e capturar com sucesso todas as ações esperadas para cada um dos programas especialmente desenvolvidos, cuja execução é formada por ações conhecidas e comuns em processos maliciosos. Mesmo em casos no qual *packers* que inviabilizam a análise em ambiente emulado estavam presentes, BehEMOT exibiu o resultado correto em 100% deles. Dos seis aplicativos criados a partir de ações comumente encontradas em *malware*, BehEMOT produziu relatórios de análise que compreendem todas as ações esperadas para tais exemplares, as quais variaram desde a criação de arquivos e registros até tráfego de rede.

Os testes comparativos com outras ferramentas de análise existentes, as quais se utilizam de técnicas diferentes de monitoração de comportamento, mostraram que BehEMOT se comporta de maneira eficaz, capturando informações não capturadas por outras ferramentas . Vale ressaltar que não existe uma referência que indique qual análise está correta, visto que não é possível saber ao certo o comportamento esperado de um certo *malware*. Para contornar este problema, foi convencionado na métrica de comparação que o relatório maior, com mais ações capturadas durante a execução, é a referência, representando portanto um dos possíveis comportamentos apresentados pelo *malware*. Usou-se esta convenção pois o comportamento menor, que contêm menos ações, normalmente é referente a uma execução incompleta, que terminou com algum erro devido alguma técnica de evasão utilizada pelo *malware* para ocultar seu comportamento ou limitação do sistema de análise, que terminou a execução do *malware* de forma inesperada.

Nos casos onde a comparação resultou em baixas taxas de inclusão, o relatório produzido por BehEMOT foi capaz de mostrar as ações desenvolvidas por todos os exemplares de *malware* analisados, o que não ocorreu com as outras ferramentas da comparação, *Anubis* e, principalmente, *CWSandbox*. Esta última teve uma grande parcela de exemplares de *malware* não analisados, mesmo com repetidas submissões. Os altos índices de baixa taxa de inclusão entre os relatórios de BehEMOT e de *CWSandbox* são devidos principalmente à diferença na quantidade de atividades capturadas por ambos, sendo que BehEMOT captura um maior volume de informações úteis na maioria dos casos. Os relatórios provenientes de análises realizadas por *CWSandbox* contém muita informação, o que dificulta a interpretação do comportamento malicioso apresentado pelo *malware*. Nas análises realizadas em BehEMOT, estes tipos de ações são filtradas, revelando somente

[1] *System Service Dispatch Table*

os dados que apresentam concordância com o comportamento do *malware*. Em relação a *Anubis*, a comparação com BehEMOT resultou em maior quantidade acumulada de relatórios com melhor taxa de inclusão, uma vez que os relatórios de ambas as ferramentas capturam em grande parte dos casos o mesmo tipo e quantidade de informação.

Já nos testes com *malware* que têm *packers* do tipo *tElock!*, cuja principal característica é evitar a execução em emuladores, os resultados produzidos por BehEMOT são melhores do que os produzidos por *Anubis*. O resultado produzido por Anubis, que mostra apenas a parte do comportamento de execução que antecede à evasão da análise devido ao uso do *tElock!*, foi ineficaz para exemplares de malware com este tipo de packer. Por outro lado, *CWSandbox* também se mostrou eficiente na análise destes tipos de *malware*, porém, em alguns casos não foi produzido o mesmo volume de informação sobre a execução do *malware* em comparação a BehEMOT.

5.1 Trabalhos Futuros

Por fim, apesar de BehEMOT mostrar-se como uma alternativa viável para a análise dinâmica de *malware*, superando eventuais problemas com *packers* ou outros possíveis mecanismos de evasão de emuladores ou máquinas virtuais, existem algumas limitações que permitem sua detecção e consequente evasão. A capacidade de tratar as características que causam tais limitações a fim de minimizar a detecção é deixada como trabalho futuro. Os agentes que podem causar a detecção dentro do ambiente de análise são discutidos a seguir.

O controlador e o *driver*, componentes principais de BehEMOT, estão presentes dentro ambiente de análise e podem ser encontrados por exemplares *malware* que especificamente procurem por este tipo de informação, fazendo com que este finalize sua execução, esconda seu comportamento malicioso e torne a análise incorreta. Entretanto, é possível realizar determinadas modificações em BehEMOT de forma a ocultar seus componentes, o que torna a detecção do ambiente de análise não-trivial. Isto pode ser feito através do próprio *driver*, ao se empregar certas características presentes em *rootkits*.

Outra limitação presente em BehEMOT é a sua dependência com a versão do sistema operacional utilizado no ambiente de análise. A técnica de *SSDT hooking* precisa ser revista sempre que uma nova atualização ao *kernel* do sistema (por exemplo, alguns *Service Packs*) é divulgada, dado que a tabela que contém os endereços das *system calls* pode ser modificada por este processo. Uma possível solução envolve a inclusão de uma verificação para cada uma das versões existentes de sistemas *Windows*, de forma que quando o *Driver* de monitoração for instalado no ambiente de análise o *SSDT hooking* seja feito de forma correta. O exemplar de *malware* também pode verificar a *SSDT* em busca de alterações, identificando um possível ambiente de análise por meio da presença de *hooks*. A solução

deste problema pode ser implementada trocando-se a técnica, através da utilização de *Inline Hooking*, a qual mantém os endereços originais da *SSDT*, modificando apenas o código *assembly* presente no início de cada *system call*.

5.2 Publicações

Durante o desenvolvimento deste trabalho, foram geradas 3 publicações que envolvem o trabalho apresentado. As publicações [54] e [55] abordam diretamente a arquitetura do protótipo BehEMOT, sem os testes comparatívos. Em [56], o componente de captura do BehEMOT foi utilizado para extrair as ações executadas pelo processo do *Internet Explorer*, para identificar possíveis ações maliciosas realizadas por ele.

Referências Bibliográficas

[1] Vinod P. and M.S.Gaur V.Laxmi. Survey on Malware Detection Methods. http://www.security.iitk.ac.in/contents/events/workshops/iitkhack09/papers/vinod.pdf.

[2] McAffe Labs. Mcafee Threats Report: Third Quarter 2010. Technical report, McAfee, 2010. http://www.mcafee.com/us/local_content/reports/q32010_threats_report_en.pdf.

[3] Wei Yan, Zheng Zhang, and Nirwan Ansari. Revealing Packed Malware. *IEEE Security and Privacy*, 6(5):65–69, 2008.

[4] Symantec Global Internet Security Threat Report – Trends for 2009. Technical Report Volume XV, Symantec security response, April 2010. http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_internet_security_threat_report_xv_04-2010.en-us.pdf.

[5] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 289–300, Washington, DC, USA, 2006. IEEE Computer Society.

[6] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. OmniUnpack: Fast, Generic, and Safe Unpacking of Malware. In *ACSAC*, pages 431–441. IEEE Computer Society, 2007.

[7] V. Sai Sathyanarayan, Pankaj Kohli, and Bezawada Bruhadeshwar. Signature Generation and Detection of Malware Families. In *Proceedings of the 13th Australasian conference on Information Security and Privacy*, pages 336–349, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] Yangseo Choi, Jintae Oh, Jeonggun Lee, and Jaecheol Ryou. Optimal position searching for automated malware signature generation. In *Consumer Electronics, 2009. ISCE '09. IEEE 13th International Symposium*, pages 561–564, 2009.

[9] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: a hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode*, WORM '07, pages 46–53, New York, NY, USA, 2007. ACM.

[10] Monirul I. Sharif, Vinod Yegneswaran, Hassen Saïdi, Phillip A. Porras, and Wenke Lee. Eureka: A Framework for Enabling Static Malware Analysis. In Sushil Jajodia and Javier López, editors, *ESORICS*, volume 5283 of *Lecture Notes in Computer Science*, pages 481–500. Springer, 2008.

[11] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of Static Analysis for Malware Detection. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 421 –430, dec. 2007.

[12] Timothy E. Levin, Cynthia E. Irvine, Clark Weissman, and Thuy D. Nguyen. Analysis of three multilevel security architectures. In *Proceedings of the 2007 ACM workshop on Computer security architecture*, CSAW '07, pages 37–46, New York, NY, USA, 2007. ACM.

[13] Mark E. Russinovich and David A. Solomon. *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server(TM) 2003, Windows XP, and Windows 2000 (Pro-Developer)*. Microsoft Press, Redmond, WA, USA, 2004.

[14] Bill Blunden. *The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System*. Jones and Bartlett Publishers, Inc., USA, 2009.

[15] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A Survey on Automated Dynamic Malware Analysis Techniques and Tools. ACM Computing Surveys Journal, to appear.

[16] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[17] Vmware, Julho 2011. `http://www.vmware.com/`.

[18] Virtualbox, Julho 2011. `http://www.virtualbox.org/`.

[19] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing CPU emulators. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, pages 261–272, New York, NY, USA, 2009. ACM.

[20] Mendel Rosenblum. The Reincarnation of Virtual Machines. *Queue*, 2:34–40, July 2004.

[21] Greg Hoglund and Jamie Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005.

[22] Anubis - Analyzing Unknown Binaries, Março 2011. `http://anubis.iseclab.org/`.

[23] Ulrich Bayer, Christopher Kruegel, and Engin Kirda. TTanalyze: A Tool for Analyzing Malware, 2006.

[24] Thomas Raffetseder, Christopher Krugel, and Engin Kirda. Detecting System Emulators. In *ISC*, pages 1–18, 2007.

[25] Danny Quist and Val Smith. Detecting the Presence of Virtual Machines Using the Local Data Table. `http://www.offensivecomputing.net/files/active/0/vm.pdf`.

[26] Min Gyung Kang, Heng Yin, Steve Hanna, Stephen McCamant, and Dawn Song. Emulating emulation-resistant malware. In *Proceedings of the 1st ACM workshop on Virtual machine security*, VMSec '09, pages 11–22, New York, NY, USA, 2009. ACM.

[27] Tom Liston and Ed Skoudis. On the Cutting Edge: Thwarting Virtual Machine Detection., 2006. `http://handlers.sans.org/tliston/ThwartingVMDetection\_Liston\_Skoudis.pdf`.

[28] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 51–62, New York, NY, USA, 2008. ACM.

[29] Gábor Pék, Boldizsár Bencsáth, and Levente Buttyán. nEther: in-guest detection of out-of-the-guest malware analyzers. In *Proceedings of the Fourth European Workshop on System Security*, EUROSEC '11, pages 3:1–3:6, New York, NY, USA, 2011. ACM.

[30] Holy Father. Hooking Windows API—Technics of Hooking API Functions on Windows, 2004.

[31] Microsoft Portable Executable and Common Object File Format Specification. `http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx`.

[32] Galen Hunt and Doug Brubacher. Detours: binary interception of Win32 functions. In *Proceedings of the 3rd conference on USENIX Windows NT Symposium - Volume 3*, pages 14–14, Berkeley, CA, USA, 1999. USENIX Association.

[33] Cuckoo Sandbox - Automated Malware Analysis System, Março 2011. `http://www.cuckoobox.org/`.

[34] CWSandbox :: Behavior-based Malware Analysis, Julho 2011. `http://mwanalysis.org/`.

[35] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security and Privacy*, 5(2):32–39, March 2007.

[36] JoeBox, Julho 2011. `http://www.joesecurity.org/`.

[37] Christian Seifert, Ramon Steenson, Ian Welch, Peter Komisarczuk, and Barbara Endicott-Popovsky. Capture – a behavioral analysis tool for applications and documents. In *The International Journal of Digital Forensics Incident Response*. Elseview, 2007.

[38] CaptureBat, Março 2011. `http://www.honeynet.org/project/CaptureBAT`.

[39] Estatística sobre uso de sistemas operacionais., Abril 2011. `http://www.w3schools.com/browsers/browsers_os.asp`.

[40] Mohammed Abdul Qadeer, Arshad Iqbal, Mohammad Zahid, and Misbahur Rahman Siddiqui. Network Traffic Analysis and Intrusion Detection Using Packet Sniffer. *Communication Software and Networks, International Conference on*, 0:313–317, 2010.

[41] Trojan.Dropper, Março 2011. `http://www.symantec.com/security\_response/writeup.jsp?docid=2002-082718-3007-99`.

[42] Ulrich Bayer, Imam Habibi, Davide Balzarotti, Engin Kirda, and Christopher Kruegel. A view on current malware behavior. In *LEET 09: 2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats*, 04 2009.

[43] Snapshot technology overview, Abril 2011. `http://www.ibm.com/developerworks/tivoli/library/t-snaptsm1/index.html`.

[44] Partimage, Abril 2011. `http://www.partimage.org/Main\_Page`.

[45] Grub - Multiboot boot loader., Abril 2011. `http://www.gnu.org/software/grub/`.

[46] Walter Oney. *Programming the Microsoft Windows Driver Model, Second Edition*. Microsoft Press, Redmond, WA, USA, 2nd edition, 2002.

[47] Mark E. Russinovich, David A. Solomon, and Alex Ionescu. *Windows internals*. Microsoft Press Series. Microsoft Press, 2009.

[48] Seungwon Shin and Guofei Gu. Conficker and beyond: a large-scale empirical study. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 151–160, New York, NY, USA, 2010. ACM.

[49] Felix Leder and Tillmann Werner. Know Your Enemy: Containing Conficker, To Tame a Malware. Technical report, The Honeynet Project, http://honeynet.org, April 2009.

[50] PEiD - Portable Executable Identifier, Abril 2011. `http://www.peid.info/`.

[51] TCPDUMP packet analyzer., Abril 2011. `http://www.tcpdump.org/`.

[52] VX Heavens, Abril 2011. `http://vx.netlux.org/`.

[53] Dionaea, Abril 2011. `http://dionaea.carnivore.it/`.

[54] D.S. Fernandes Filho, A.R.A. Grégio, V.M. Afonso, R.D.C. Santos, M. Jino, and P.L. de Geus. Análise comportamental de código malicioso através da monitoração de chamadas de sistema e tráfego de rede. In *Anais do X Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*, pages 311–324, 2010.

[55] A.R.A. Grégio, D.S. Fernandes Filho, V.M., R.D.C. Santos, M. Jino, and P.L. de Geus. Behavioral analysis of malicious code through network traffic and system call monitoring. volume 8059, page 80590O. SPIE, 2011.

[56] V.M. Afonso, A.R.A. Grégio, D.S. Fernandes Filho, and P.L. de Geus. A hybrid system for analysis and detection of web-based client-side malicious code. In *Proceedings of the IADIS International Conference WWW/Internet 2011*, 2011.