Análise Transparente de *Malware* com Suporte por *Hardware*

Marcus Botacin[1], **Paulo Lício de Geus**[1], **André Grégio**[2]

[1]Instituto de Computação (IC)
Universidade Estadual de Campinas (Unicamp)
Campinas – SP – Brasil

[2]Departamento de Informática (DInf)
Universidade Federal do Paraná (UFPR)
Curitiba – PR – Brasil

Abstract. *Dynamic analysis is one of the main techniques used for malware profiling, identification of features and development of countermeasures. Therefore, malware authors continuously seek for ways of preventing their code from running inside analysis environments to thwart detection. Besides, operating system improvements make their instrumenting for malware monitoring more difficult. Hence, hardware-assisted analysis approaches have been developed to overcome these issues. In this paper, we propose a low-overhead malware dynamic analysis system based on branch monitoring supported by hardware (Intel processor monitors), in order to accomplish the transparency required to prevent malware from identifying (and evading) monitoring.*

Resumo. *A análise dinâmica é uma das principais técnicas utilizadas para caracterização de malware, identificação de suas funcionalidades e desenvolvimento de contra-medidas. Portanto, desenvolvedores de malware buscam continuamente por formas de impedir a execução de seus códigos nesses ambientes, dificultando a detecção. Além disso, avanços nos sistemas operacionais criaram obstáculos para sua instrumentação. Para superar tais problemas, surgiram abordagens de análise assistidas por hardware. Neste artigo, propõe-se um sistema de análise dinâmica de malware baseado em monitoramento de branches com suporte de hardware (monitores dos processadores Intel), alcançando a transparência necessária para evitar a identificação do ambiente e apresentando baixo overhead.*

1. Introdução

Os danos causados por programas maliciosos abrangem desde o comprometimento de um sistema operacional até prejuízos pessoais e financeiros aos usuários. Estatísticas do CERT.br [CERT.br 2015] indicam que mais de 50% dos incidentes reportados, como *scans* e fraudes, podem ter sido causados por *malware*. Perdas devido a fraudes eletrônicas alcançam 1,8 bilhão de Reais [FEBRABAN 2015]. A análise de *malware* é uma etapa importante para a identificação do potencial de danos e extensão destes, bem como para o desenvolvimento de vacinas, assinaturas de detecção e procedimentos de perícia forense computacional. As técnicas de análise são classificadas como estáticas ou dinâmicas [Sikorski and Honig 2012].

Na análise estática, o arquivo suspeito é inspecionado sem sua execução, através apenas da observação de suas características estruturais (*headers* e *strings*), estatísticas

(entropia) e *disassembly*. Entretanto, limitações intrínsecas [Moser et al. 2007] trazem a necessidade adicional da análise dinâmica na qual o exemplar é efetivamente executado em um ambiente controlado (*sandbox*), exibindo seu comportamento de infecção. Contudo, criadores de *malware* têm buscado formas de identificar a execução do exemplar em um ambiente de análise [Chen et al. 2008] de modo a bloquear a execução do *payload* malicioso e evadir a detecção do *malware*. Dentre as técnicas utilizadas para tal fim, destacam-se as baseadas na detecção de emulação e de injeção de código.

Diante deste novo cenário, muitas técnicas foram propostas para mascarar os efeitos colaterais da monitoração, usados pelos atacantes para a identificação do ambiente de análise [Balzarotti et al. 2010, Vasudevan and Yerraballi 2005]. Porém, qualquer abordagem via *software* não está protegida o suficiente da subversão por parte do atacante. Assim sendo, abordagens com suporte de *hardware* têm sido propostas. As mais bem sucedidas delas baseiam-se na criação de uma máquina virtual de análise que usa instruções especiais do processador (*Hardware Virtual Machine* - HVM) e na reescrita do *Basic Input Output System - BIOS* para analisar a execução a partir de um modo privilegiado (*System Management Mode* - SMM). Embora eficazes, dado sua *Trusted Code Base* (TCB) reduzida, as abordagens citadas apresentam alto custo de desenvolvimento e desempenho, com *overhead* acima de 100% dependendo da frequência de interrupção.

Recentemente, uma abordagem assistida por *hardware* menos custosa (*lightweight*) surgiu através do uso de monitores de desempenho, sobretudo os monitores de desvios (*branches*), os quais possuem *overhead* próximo de zero. O uso atual desses monitores tem sido a detecção de ataques por *Return Oriented Programming* (ROP). Neste trabalho, introduz-se uma nova aplicação para estes monitores e importante contribuição para a área de segurança: a geração de traços de execução de *malware* de forma transparente (na visão do programa monitorado). Até onde se sabe, este é o primeiro trabalho a fazer uso deste monitores para tal propósito. Ao longo do artigo, detalha-se as vantagens e limitações da abordagem proposta.

Este artigo é dividido da seguinte maneira: na Seção 2, introduz-se os conceitos e os detalhes de funcionamento dos monitores utilizados, as abordagens existentes para monitoramento de *malware* e as limitações destas; na Seção 3, apresenta-se o sistema proposto, incluindo detalhes de sua implementação; na Seção 4, aplica-se a solução implementada na reconstrução do *Call Graph* e do *Control Flow Graph* de um código compilado, bem como a exemplares reais de *malware*; na Seção 5, discute-se os desafios, o desempenho e os limites do sistema proposto; por fim, a Seção 6 apresenta as considerações finais sobre os resultados obtidos e possíveis evoluções para o sistema.

2. Aspectos Técnicos e Trabalhos Relacionados

Descreve-se, nesta seção, os detalhes técnicos relativos às plataformas de monitoração usando *hardware*. Embora presentes em diversas arquiteturas, limita-se a descrição à tecnologia utilizada pela Intel, escolhida para a implementação do protótipo deste trabalho por questões de disponibilidade. Apresenta-se também um panorama das abordagens para monitoração de *malware*, agrupadas por modo de atuação, de modo que se possa avaliar as limitações de cada uma delas a fim de justificar a escolha dos monitores de desempenho.

2.1. Monitoração assistida por *hardware*

A plataforma de monitoração da Intel pode ser dividida em dois componentes, o *Precise Event Based Sampling* (PEBS) e o monitoramento de *branches*. O PEBS consiste em uma série de registradores configuráveis para contar diferentes eventos, tais como instruções executadas, número de ciclos transcorridos, *misses* de memória cache, entre outros. Os dados gerados pelo PEBS podem ser acessados via registradores específicos (*Model Specific Register* - MSR) ou registro em memória. Neste último, uma página do sistema operacional pode ser fornecida para o armazenamento dos dados, gerando uma interrupção quando o *buffer* estiver cheio. O monitoramento de *branches* atua de forma similar, mas com escopo nas informações dos endereços de origem e destino das instruções de desvio, como mostrado na Figura 1. O mecanismo fornece ainda, para cada desvio, a informação se o salto foi ou não previsto corretamente (*branch-prediction*).

FROM	TO
fff80145f42118	fff8801ae01f36
fff80145e5a510	fff80145ecf2ae
fff80145f420dc	ff80146079523
fff80145ecf2b3	ffff8801ae0157

Figura 1. Exemplo de *branch stack* obtida com o mecanismo de monitoramento do processador.

Os dados obtidos podem ser armazenados de duas formas: a primeira, denominada *Last Branch Record* (LBR), consiste em pares de registradores (de 8 a 32, dependendo da família de processadores) do processador que formam uma fila circular, sendo acessível através dos registradores MSR; a segunda, denominada *Branch Trace Store* (BTS), consiste em um mecanismo análogo ao PEBS de fornecimento de páginas do sistema operacional para o armazenamento dos dados, podendo gerar uma interrupção quando o *buffer* estiver cheio. Embora denominado monitor de *branch*, os dados capturados não se restringem às instruções de salto do processador (`JMP`, `JNE`), mas a qualquer desvio do fluxo, incluindo instruções `CALL` e `RET`. A captura de dados de cada um dos tipos de instruções de desvio pode ser filtrada pela inserção de *flags* em um registrador de controle. Pode-se ainda filtrar ações nos níveis de *kernel* e *userland* e decidir se a captura será ou não interrompida quando uma interrupção for gerada.

2.2. Trabalhos Relacionados

Os trabalhos relacionados discutidos a seguir são divididos de acordo com a técnica utilizada para monitoração do *malware* em execução.

Análise em *userland*. A monitoração no nível do usuário é uma das formas mais simples de implementação para interceptação de programas em execução. A inserção de

hooks via injeção de *DLL* permite interceptar qualquer ação de determinado processo, sendo usada pelos sistemas de análise de *malware* Cuckoo Sandbox [Guarnieri 2013] e CWSandbox [Willems et al. 2007]. Porém, além de requerer que a injeção da DLL de monitoração seja realizada em cada processo sob análise, a técnica é facilmente detectável, pois altera a imagem do processo em memória.

Tradução dinâmica. Algumas ferramentas atuam em nível mais granular do que o das chamadas de sistema, podendo lidar diretamente com as instruções do binário, tais como Valgrind [Nethercote and Seward 2003], PIN [Luk et al. 2005] e DynamoRIO [Bruening et al. 2012]. Este tipo de abordagem traz mais possibilidades para detecção, criação de heurísticas e demais formas de monitoração, mas pode ser evitada pelo *malware* dados seus efeitos colaterais—comportar-se como um *debugger* ou alterar as instruções do programa. Outra geração de instrumentadores dinâmicos surgiu mais recentemente, como VAMPIRE [Vasudevan and Yerraballi 2005] e SPIKE [Vasudevan and Yerraballi 2006], capazes de alterar as instruções de evasão em tempo real. Porém, esse tipo de solução depende de regras pré-estabelecidas que podem ser subvertidas por novas técnicas de evasão implementadas por desenvolvedores de *malware*.

***Hardware-assisted Virtual Machines* (HVM).** Dada as limitações intrínsecas das abordagens por software, a evolução natural é obter suporte via *hardware*, cuja subversão é mais difícil. Um exemplo são as máquinas virtuais em *hardware* (HVM), nas quais as instruções de virtualização do processador são aplicadas na construção de um sistema de análise. Ether [Dinaburg et al. 2008], MAVMM [Nguyen et al. 2009], CXPInspector [Carsten Willems 2012] e SPIDER [Deng et al. 2013] são exemplos de sistemas baseados em HVM. Tais sistemas possuem características singulares: implementação minimalista, sem *drivers* e componentes adicionais, reduzindo assim a base de código confiável (*Trusted Code Base* - TCB) requerida e tornando-se menos "exploráveis"; ausência de efeitos colaterais, pois as instruções não são emuladas ou traduzidas, mas sim executadas no *hardware* real; granularidade da análise, uma vez que qualquer subsistema pode ser monitorado diretamente. Por outro lado, sistemas baseados em HVM possuem desvantagens associadas: são muito complexos de implementar, devido à necessidade de escrita de um *hypervisor*; limitados no uso prático, por exemplo, devido à restrição a execução em *single-core* e/ou a exigência de *hypervisor* específico; e são dependentes da versão e do sistema analisado, pois a instrospecção é baseada nos *offsets* e estruturas de dados opacas de cada *kernel*.

***System Management Mode* (SMM).** Outra abordagem com auxílio de *hardware* é a instrumentação das rotinas do modo SMM dos processadores. Trata-se de um modo protegido, com memória isolada e voltado para o gerenciamento do sistema, o qual possui funções para controle de energia e temperatura. O isolamento provido por este modo garante o não comprometimento trivial do sistema de análise. A análise realizada neste modo é transparente—as instruções são executadas diretamente no hardware, sem qualquer tradução ou emulação—e as ferramentas baseadas em SMM visam tanto *debugging* (MALT [Zhang et al. 2015] e SPECTRE [Zhang et al. 2013]) quanto forense do sistema operacional (SMMDumper [Reina et al. 2012]). Assim como as abordagens de HVM, a escrita de uma ferramente SMM é complexa, exigindo técnicas de introspecção e a reescrita do BIOS, a qual nem sempre é possível devido a travas que visam garantir sua

integridade. O modo SMM também exige a escrita de *drivers* de dispositivos, como os de rede, para que se possa enviar os dados da máquina monitorada para o meio externo.

***Performance Counters*.** As restrições inerentes às soluções supracitadas incentivaram o surgimento de outras ferramentas—menos custosas em processamento e implementação—para o monitoramento assistido por *hardware*. A que mais se destaca e está presente na maioria das arquiteturas atuais é representada pelos monitores de desempenho, que permite coletar dados de *caches*, instruções e *branches*. Seu uso é bastante difundido em ferramentas de *profiling* como o `perf`[1] e o *Intel V-Tune* [Intel 2015]. Porém, as aplicações em segurança computacional são recentes: [Kompalli 2014] mostra como os monitores de *branch* e de uso de memória podem indicar comportamento anômalo em um sistema; *Kbouncer* [Pappas et al. 2013] e *Ropecker* [Cheng et al. 2014] são usadas na detecção de ataques por ROP.

Uma das maiores limitações das abordagens existentes é a forma de coleta de dados, por vezes feita de maneira *system-wide*, sem monitoramento individual de processos e de forma intrusiva (injeção de código). Ressalta-se ainda que nenhuma delas monitora a interação do *malware* com o sistema, mas sim seu fluxo de execução. Portanto, superar os desafios atuais é a maior contribuição do presente artigo.

3. Projeto do Sistema Proposto

Nesta seção, mostra-se as escolhas de projeto e os detalhes de implementação do sistema proposto para análise transparente assistida por *hardware* de programas maliciosos.

3.1. Modelo de Ameaças e Decisões Tomadas

Assume-se que os exemplares sob análise possuem as seguintes características: atuam no nível do usuário, isto é, não carregam *drivers* no sistema operacional alvo; podem ser equipados com técnicas de anti-análise; e interagem com o sistema operacional alvo por meio de suas chamadas de APIs. Este modelo de ameaça embasa uma série de decisões de projeto, apresentadas abaixo.

- **Análise em *userland***: a coleta de informações dos mecanismos de monitoramento só pode ser feita em modo *kernel*, seja pela necessidade de se lidar com páginas do sistema, seja pelo acesso a registradores especiais (MSR). De modo geral, um *driver* é responsável por essa tarefa. A garantia de que exemplares analisados estarão em *userland* elimina a possibilidade de subversão do mecanismo de análise, já que este estará em um nível mais privilegiado [Rossow et al. 2012].
- **Análise transparente**: as ferramentas com suporte de *hardware* exigem ambiente real (*bare-metal*) para a sua execução, fazendo que muitas técnicas de anti-análise falhem, sobretudo as baseadas em detecção de máquina virtual. Contudo, outras técnicas, como a verificação da presença de *debuggers*, ainda poderiam ser efetivas em evitar a análise. Os mecanismos PEBS e LBR/BTS são habilitados via registradores de controle que também controlam o uso de registradores de *debug*, e a má configuração destes poderia permitir a identificação de seu uso. Para evitar este cenário, não foram utilizadas *flags* que habilitassem o uso de registradores de *debug*, como no caso de *single-step on branch*.

[1] https://perf.wiki.kernel.org/index.php/Main_Page

- **Uso de APIs de sistema**: considerando que o mecanismo de *branch* monitora endereços-alvo, se estes forem conhecidos de antemão, pode-se relacionar o alvo do *branch* com a biblioteca carregada no referido endereço. Se os exemplares usarem o mecanismo de chamadas de sistema para sua execução, garante-se a reconstrução de todo o fluxo de chamadas destes.
- **Sistema operacional moderno**: a partir do Vista, sistemas operacionais da família Windows apresentam restrições às modificações que podem ser feitas no *kernel* para fins de monitoração. Técnicas alternativas são, em geral, baseadas em filtros limitados à monitoração de eventos providos por uma interface do SO. O sistema proposto monitora qualquer atividade no sistema, independentemente dos meios providos por este para tal fim. Para tanto, combinou-se o mecanismo de monitoração de *branch* com uma técnica de introspecção.

3.2. Implementação

Nesta seção, os detalhes específicos de implementação do sistema proposto são apresentados, considerando-se aspectos de funcionamento e interação com o sistema operacional alvo. O protótipo para testes foi desenvolvido em Windows 8 de 64 bits em execução sob processador Intel Core i5 com plataforma *Haswell*.

3.2.1. Captura de dados

Optou-se então por utilizar o modo BTS, que permite a gravação em páginas do sistema e a geração de uma interrupção para alertar quando o *buffer* está cheio. Além de mais de 16 endereços/registros (limite do modo LBR na geração atual de processadores, incorrendo em captura insuficiente e ineficiente de dados), pode-se capturar os dados sem o risco de perder qualquer instrução, o que ocorreria se a implementação fosse feita por *polling*. A interrupção gerada garante que os processos do sistema operacional encontram-se suspensos no momento da captura, permitindo a inspeção destes sem efeitos colaterais.

Um problema existente nas demais soluções de monitoramento é a não identificação da origem dos saltos. Para resolvê-lo, configurou-se o *threshold* de interrupção para apenas uma posição, ou seja, interrompe-se o fluxo a cada instrução de desvio para se obter o último processo em execução (gerador do desvio e, portanto, origem). O funcionamento do mecanismo de interrupção consiste no processador chamar a *Interrupt Service Routine* registrada na *Interrupt Descriptor Table* (IDT) na posição dada pelo vetor definido na *Advanced Programmable Interrupt Controller - Local Vector Table* (APIC-LVT).

Tal interrupção deveria, a princípio, ser entregue em modo *Fixed-Edge Delivery*. Contudo, o Windows já registra uma interrupção para seu próprio uso na faixa de valores do monitor e a alteração dessa rotina é impedida pelo mecanismo de proteção de *kernel*[2]. As alternativas encontradas incluem a desativação deste mecanismo e a utilização de funções não documentadas. Porém, visando minimizar qualquer subversão do sistema operacional alvo e tornar o protótipo mais portável, optou-se por alterar o modo de entrega da interrupção para *Non-Maskable Interrupt* (NMI)—uma interrupção especial para tratar exceções e erros cujo *handler* é nativo do sistema operacional.

[2]`http://technet.microsoft.com/pt-br/library/cc759759(v=ws.10).aspx`

3.2.2. Funcionamento do Sistema

Quando uma instrução de desvio é executada, o processador preenche o *buffer* fornecido com os dados referentes a ela e lança a interrupção. Esse monitoramento é totalmente executado por *hardware* e não depende de qualquer injeção de código no processo monitorado. Na prática, todos os processos são monitorados, já que o processador não é ciente de qual processo gerou aquele *branch*. A identificação do processo é realizada pelo tratador da interrupção através da função `PsGetCurrentProcess`[3];

A interrupção é tratada em *kernel* por meio de um *driver*, o qual é responsável por ler os valores do *buffer* e colocá-los em uma fila na ordem em que as interrupções foram geradas, para posterior análise, atuando como o servidor de uma arquitetura cliente-servidor. Nessa arquitetura, o cliente é uma aplicação independente em modo usuário que recebe os valores lidos pelo *driver*. O cliente também responde por todo o processo de coleta, introspecção e visualização de dados, além de filtrar e exibir informações apenas dos processos desejados, uma vez que as informações recebidas incluem dados de todo o sistema.

Essa abordagem "*system-wide*" deve-se aos exemplares de *malware* atuais serem desesenvolvidos modularmente e obterem funcionalidades adicionais por *download* de componentes, frequentemente dividindo a infecção entre diferentes processos. Embora através da introspecção seja possível interceptar a chamada de API da criação de processos, não é possível obter diretamente o PID do processo filho criado pelo exemplar. Para resolver esse problema, registrou-se uma *callback* de processos[4] para a identificação do processo-filho, permitindo ao cliente adicionar tal PID à lista de processos monitorados.

3.2.3. Introspecção

A tradução dos endereços-alvo dos desvios para informações de mais alto nível é um processo conhecido como introspecção. Através deste processo é possível, por exemplo, reconstruir o grafo das chamadas de função de um exemplar de *malware* (*call graph*). O primeiro passo da introspecção é descobrir o endereço-base das bibliotecas. Em sistemas operacionais modernos, os endereços de carregamento são aleatorizados a cada execução para dificultar ataques de injeção de código. A Tabela 1 mostra o endereço-base de bibliotecas em duas inicializações do sistema.

Tabela 1. Exemplo do efeito do mecanismo de aleatorização de endereços (ASLR) sobre os módulos dinâmicos em duas inicializações consecutivas do SO.

Biblioteca	ntdll.dll	KERNEL32.DLL	KERNELBASE.dll	NETAPI32.dll
Endereço 1	0xBAF80000	0xB9610000	0xB8190000	0xB6030000
Endereço 2	0x987B0000	0x98670000	0x958C0000	0x93890000

A solução encontrada para analisar *malware* adequadamente apesar do mecanismo de ASLR é obter os endereços das bibliotecas a cada reinicialização do sistema, através

[3]https://msdn.microsoft.com/en-us/library/windows/hardware/ff559933%28v=vs.85%29.aspx
[4]https://msdn.microsoft.com/en-us/library/windows/hardware/ff542860%28v=vs.85%29.aspx

da função `GetModuleHandle`[5]. A partir do endereço-base das bibliotecas, é preciso identificar qual das funções foi chamada, pois cada biblioteca pode conter diferentes funções. A Tabela 2 mostra diferentes funções da biblioteca `ntdll.dll`. Com tais endereços, a introspecção é então feita da seguinte forma: dado um endereço, obtém-se o endereço-base de uma biblioteca conhecida mais próxima, subtrai-se este do valor original e considera-se o valor resultante como *offset*. O processo descrito é ilustrado pela Figura 2.

Tabela 2. Exemplos de *offsets* das funções de biblioteca `ntdll.dll`

Função	***Offset***
NtCreateProcess	0x3691
NtCreateProcessEx	0x30B0
NtCreateProfile	0x36A1
NtCreateProfileEx	0x36B1
NtCreateResourceManager	0x36C1
NtCreateSemaphore	0x36D1
NtCreateSymbolicLinkObject	0x36E1
NtCreateThread	0x30C0
NtCreateThreadEx	0x36F1

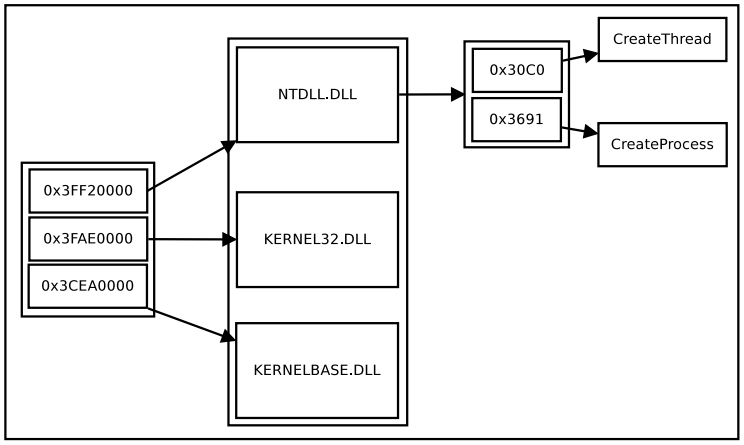

Figura 2. Diagrama de funcionamento do mecanismo de introspecção para associação de nomes de funções a endereços de módulos.

4. Aplicações, Testes e Resultados

Para validar o sistema, foram realizados testes baseados na geração de grafos de chamadas a partir da monitoração de códigos escolhidos para execução. O grafo de chamadas (*callgraph* ou CG) é um grafo onde os vértices representam funções e as arestas a relação entre elas (dependência, temporalidade etc.). Um dos usos de CG para detecção de códigos maliciosos é a identificação de variantes metamórficas com base em busca por isomorfismos [Martins et al. 2014]. No sistema proposto neste artigo, as funções chamadas são obtidas diretamente do mecanismo de monitoramento de *branch*, quando programado

[5]https://msdn.microsoft.com/pt-br/library/windows/desktop/ms683199%28v=vs.85%29.aspx

para capturar chamadas do tipo `CALL`. Como este método de captura atua de maneira ampla (*system-wide*), pode-se reconstruir o grafo incluindo tanto as chamadas diretas de funções (*step-over*) somente, quanto as chamadas internas a estas (*step-into*), ambos discutidos adiante. Destaca-se ainda que a obtenção do CG se dá a partir do comportamento observado durante a execução e não por *disassembly*. Logo, o sistema não se confunde na presença de *dead-code*[6]. Os resultados foram alcançados a partir da execução do código exibido pela Listagem 1 (executado no sistema pelo processo `NewToy.exe`).

Listagem 1. Código de exemplo para a reconstrução do CG.

```
1  scanf("%d",&n);
2  scanf("%s",val);
3  for(i=0;i<n;i++)
4    printf("%s\n",val);
```

4.1. *Step-Into*

Neste modo, considera-se que o analista "mergulha" em cada uma das bibliotecas e funções externas chamadas, podendo observar com alta granularidade aspectos internos da implementação, resultando em um CG ampliado, como mostrado na Figura 3.

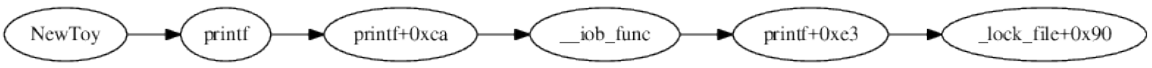

Figura 3. Visualização completa de trecho do CG.

4.2. *Step-Over*

Neste modo, o analista pode verificar as funções chamadas pelo programa sob monitoração, sem visualizar aspectos internos das bibliotecas e funções externas. Como o objetivo é ter uma visão global das chamadas, permite-se a definição de uma lista de funções de interesse, o que corresponderia à escolha das funções a serem interceptadas via *hooks* em sistemas de análise dinâmica tradicionais. O processamento consiste em realizar uma busca (como uma *Breadth First Search* - BFS) nos dados obtidos, onde os vértices correspondentes às funções de interesse, marcados em um caminho a partir da raiz, são ligados diretamente (transitividade). Desta vez, a execução resultou no CG reduzido da Figura 4.

Figura 4. Visualização do CG reduzido.

4.3. Reconstrução do CFG

O grafo de fluxo de controle (*Control Flow Graph* - CFG), representa todos os caminhos que podem ser percorridos por um programa em execução. O CFG é composto por blocos de instruções, denominados blocos básicos, limitados por instruções de desvio de fluxo [Appel and Palsberg 2003]. Esta estrutura tem uma associação natural com

[6]Código parte do executável ou ligado ao mesmo que não é acionado durante uma execução do mesmo.

as informações de desvios coletadas pelo sistema aqui proposto. Dado que as instruções executadas correspondem ao nível mais granular possível de análise, a obtenção do CFG de um programa é de grande valia para a análise de *malware*. As soluções adotadas para permitir a reconstrução do CFG a partir dos dados de *branch* estão detalhadas a seguir.

***Disassembly*.** A reconstrução do CFG requer todas as instruções existentes dentro de um bloco básico. Porém, o mecanismo monitor de *branch* só fornece informações acerca da primeira (alvo do *branch*) e da última (origem do *branch*) instrução de cada bloco. Para superar esta limitação, optou-se pelo *dump*—via `ReadProcessMemory`[7]—de todo o espaço de instruções contido entre dois *branches* consecutivos, obtendo-se os códigos de instruções como os exibidos na Listagem 2. Estes foram "disassemblados" e seu resultado pode ser visto na Listagem 3.

Listagem 2. Exemplo de *buffer* de instruções obtido a partir dos endereços fornecidos pelo mecanismo BTS.

```
1 \xff\x15\x0a\x11\x00\x00\x48\x8d\x0d\x9f\x11\x00\x00
```

Listagem 3. Conversão das instruções do *buffer* para *opcodes*.

```
1 0x1000 (size=6)   call    QWORD PTR [rip+0x110a]
2 0x1006 (size=7)   lea     rcx,[rip+0x119f]
```

***Plot* do CFG.** Para visualização da reconstrução do CFG, considera-se que os vértices do grafo são os endereços coletados pelo monitor de *branch*, enriquecidos com o *disassembly*, representando as instruções geradas, enquanto que as arestas são as transições entre os endereços coletados pelo monitor. A Figura 5 exibe a reconstrução do CFG a partir do código de exemplo, exibido na Listagem 4.

Listagem 4. Código de exemplo para a reconstrução do CFG.

```
1         a=0;
2         scanf("%d",&n);
3         for(i=0;i<n;i++)
4                 if(i%2==0)
5                         a++
6                 else
7                         a—
8                 printf("%d\n",a)
```

4.4. Testes com exemplares reais

Validados os testes de reconstrução do CG e do CFG, o sistema proposto foi usado em exemplares reais de *malware* sabidamente evasivos. A identificação destes se deu através da presença de funções de anti-análise de acordo com avaliação pela ferramenta `PEframe`[8]. A efetividade da evasão foi comprovada através da tentativa de execução destes em soluções de *sandbox* tradicionais. Todos os exemplares foram coretamente analisados pela solução proposta e os detalhes dos resultados obtidos (*traces*) podem ser encontrados na respectiva página do projeto[9].

[7]https://msdn.microsoft.com/pt-br/library/windows/desktop/ms680553%28v=vs.85%29.aspx
[8]https://github.com/guelfoweb/peframe
[9]https://sites.google.com/site/branchmonitoringproject/tools/tracer/evaluation/real-malware-tests

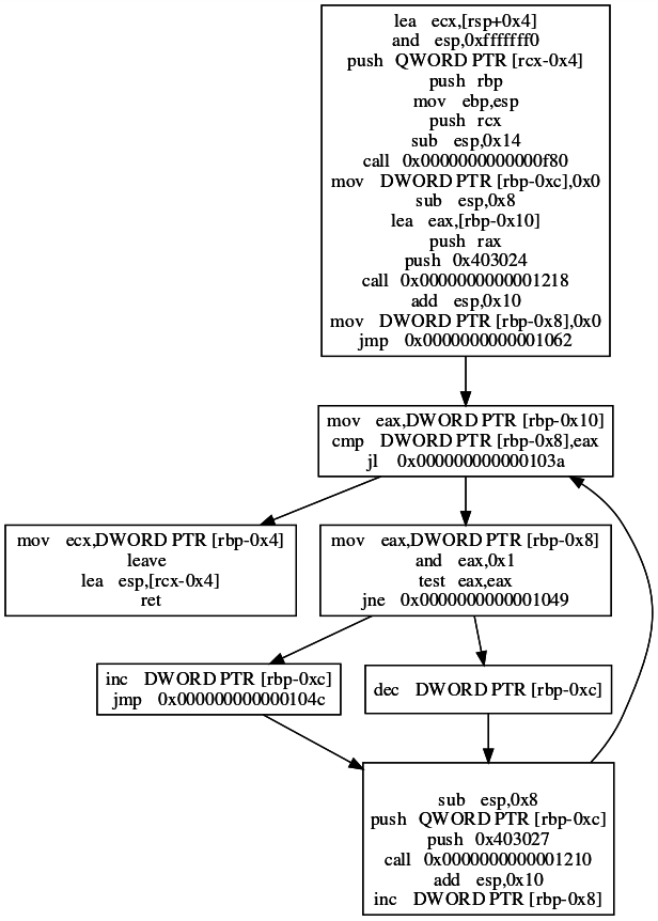

Figura 5. CFG reconstruído a partir da execução do código de exemplo.

5. Discussão

Vantagens e limitações da reconstrução do CFG. Esta abordagem, por sua natureza dinâmica, não exibe todos os caminhos possíveis de serem percorridos durante a execução do programa, o que pode ser considerado uma limitação. Entretanto, é possível assegurar a execução dos ramos exibidos no CFG (e a real instrução executada), pois o *dump* é feito nos dados coletados e não está sujeito ao desalinhamento de instruções. Ressalta-se que esta abordagem é capaz de exibir a execução de código gerado dinamicamente, pois é realizada a partir do *disassembly* do código de desvio executado pelo processador.

***Overhead* do sistema.** A detecção de sistemas de análise por meio de *timing* é uma técnica usada por autores de *malware* para impedir sua monitoração. Uma vantagem da monitoração via monitores de desempenho é o baixo *overhead*, pois trata-se de mecanismo de *hardware*—a captura dos dados por si só apresenta *overhead* teoricamente nulo. No entanto, o *overhead* total é proporcional à aplicação envolvida, sendo considerada quase nula quando se realiza apenas a exibição dos *branches* (captura), mas notável quando se realiza operações em tempo real, como o *dump* da memória. Sempre que possível, deve-se priorizar o processamento *offline* (p.ex., interpretação dos *opcodes* das instruções, que pode ser feito após a coleta dos dados de memória). Do contrário, recomenda-se a execução em um núcleo do processador diferente do monitorado, tal como na abordagem de [Quinn 2012]. Para validar tais afirmações, desenvolveu-se um programa simples

que realiza um milhão de *branches*. Comprovou-se com esse teste a nulidade prática do *overhead* imposto pela ativação tanto do mecanismo BTS quanto LBR. Verificou-se ainda que os mecanismos de interrupção e de *polling* afetam o sistema em 14% e 26%, respectivamente. O processo de introspecção impõe um custo adicional de 26% quando em execução no mesmo núcleo e zero quando em um núcleo diferente. Pôde-se verificar ainda diferenças entre os sistemas operacionais Windows e Linux. Estes resultados podem ser verificados na página do projeto[10], por questões de espaço.

Desafios. Dado o nível de abstração do monitoramento provido pelo mecanismo de monitoramento de *branch*, há desafios a serem superados. Por exemplo, a interpretação dos dados por si só apresenta desafios, pois requer grande compreensão da arquitetura sob análise (entender o funcionamento do sistema é essencial para avaliar quando os *branches* foram tomados e quando não). Além deste fator, algumas situações em especial restringem a análise. Por exemplo, a transferência da execução para o *kernel* causa perda de controle do endereçamento de retorno, uma vez que o mecanismo de BTS está desabilitado no nível deste. Embora a perda do endereço de retorno não afete nenhuma das abordagens de geração do traço de chamadas, pois elas são feitas para bibliotecas de modo usuário, tal ausência pode criar um *gap* para reconstrução do CFG caso o processo diretamente invoque *syscalls* ou faça uso de comunicação inter-processos (IPC), o que não está de acordo com o modelo de ameaça definido. A habilitação do mecanismo para captura em *kernel* pode ser abordada em trabalhos futuros, porém requer esforço adicional de interpretação—os dados precisam ser filtrados, uma vez que a proposta não considera ameaças em nível privilegiado.

Portabilidade: Embora desenvolvida na plataforma Windows, a solução pode ser aplicada nas demais plataformas e versões deste sistema, dado que o monitor em questão é um recurso do processador, e não da plataforma. O tratamento adequado dos dados deste monitor em cada plataforma exige o uso das bibliotecas compatíveis a cada uma delas.

Aplicações Futuras: A solução desenvolvida pode ser empregada para vários propósitos relativos a análise de *malware*, da identificação de novas ameaças a criação de contra-medidas. De imediato, planeja-se o agrupamento dos resultados de análise (*traces*) para o propósito de identificação de variantes evasivas.

6. Conclusão

Neste artigo, apresenta-se uma proposta de sistema para análise dinâmica de *malware* com base no suporte de *hardware* provido pela tecnologia de monitoramento de *branch* da Intel. A solução é capaz de analisar os exemplares de forma transparente, sem qualquer injeção de código e com baixo *overhead*. Os resultados obtidos mostram o funcionamento desta na reconstrução do grafo de chamadas e de fluxo de controle, permitindo análises mais granulares mesmo em sistemas operacionais modernos[11] e com restrições de *patching*. Está em andamento um trabalho baseado no sistema proposto que envolve a análise de exemplares de *malware* evasivos, isto é, equipados com técnicas de anti-forense.

[10]https://sites.google.com/site/branchmonitoringproject/tools/tracer/evaluation/overhead-measures

[11]Entende-se por modernos *kernels* do Windows a partir da versão NT 6.x de 64 bits

Agradecimentos

Os autores agradecem o apoio recebido do Conselho Nacional de Desenvolvimento Científico e Tecnológico - CNPq via Projeto MCTI/CNPq/Universal-A 14/2014 (Processo 444487/2014-0) e da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - CAPES, em especial via Projeto FORTE - Forense Digital Tempestiva e Eficiente (Processo: 23038.007604/2014-69 - Edital 24/2014 - Programa Ciências Forenses).

Referências

Appel, A. W. and Palsberg, J. (2003). *Modern Compiler Implementation in Java*. Cambridge University Press, New York, NY, USA, 2nd edition.

Balzarotti, D., Cova, M., Karlberger, C., Kruegel, C., Kirda, E., and Vigna, G. (2010). Efficient detection of split personalities in malware. In *NDSS 2010, 17th Annual Network and Distributed System Security Symposium, February 28th-March 3rd, 2010, San Diego, USA*, San Diego, UNITED STATES.

Bruening, D., Zhao, Q., and Amarasinghe, S. (2012). Transparent dynamic instrumentation. In *8th ACM SIGPLAN/SIGOPS Conf. Virtual Execution Environments*, VEE '12, pages 133–144.

Carsten Willems, Ralf Hund, T. H. (2012). Cxpinspector: Hypervisor-based, hardware-assisted system monitoring. Technical report, Horst Görtz Institute for IT Security.

CERT.br (2015). Estatísticas do cert.br. `http://www.cert.br/stats/incidentes/`. Acessado em junho/2016.

Chen, X., Andersen, J., Mao, Z. M., Bailey, M., and Nazario, J. (2008). Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 177–186.

Cheng, Y., Zhou, Z., Miao, Y., Ding, X., DENG, H., et al. (2014). Ropecker: A generic and practical approach for defending against rop attack. *Network and Distributed System Security Symposium*.

Deng, Z., Zhang, X., and Xu, D. (2013). Spider: Stealthy binary program instrumentation and debugging via hardware virtualization. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, pages 289–298, New York, NY, USA. ACM.

Dinaburg, A., Royal, P., Sharif, M., and Lee, W. (2008). 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. ACM.

FEBRABAN (2015). FEBRABAN dá dicas de segurança eletrônica. `http://www.febraban.org.br/Noticias1.asp?id_texto=2758`. Acessado em junho/2016.

Guarnieri, C. (2013). Cuckoo sandbox. `http://www.cuckoosandbox.org/`. Acessado em junho/2016.

Intel (2015). Intel vtune. `https://software.intel.com/en-us/intel-vtune-amplifier-xe`. Acessado em junho/2016.

Kompalli, S. (2014). Using existing hardware services for malware detection. In *Security and Privacy Workshops (SPW), 2014 IEEE*, pages 204–208.

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005). Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA. ACM.

Martins, G. B., Souto, E., de Freitas, R., and Feitosa, E. (2014). Estruturas virtuais e diferenciação de vértices em grafos de dependência para detecção de malware metamórfico. *Anais do SBSEG 2014*.

