Capítulo 3

Análise de binários e sistemas assistida por *hardware*

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

[1]Universidade Federal do Paraná (UFPR)
[2]Universidade Estadual de Campinas (UNICAMP)

Abstract

Binary analysis is a key step for security procedures, such as systems inspection and validation. Modern architectures are powered by many resources and features which end up in more efficient—and also more complex—applications and codes. These resources and features include virtual machine support, BIOS and chipset code execution, isolated enclaves and even external hardware. In this chapter, we introduce these features and the challenges they pose for current systems monitoring. We also present new opportunities for developing monitoring and analysis solutions based on the presented features.

Resumo

A análise de binários é uma etapa fundamental dos processos de segurança, tais como inspeção e validação de sistemas. Arquiteturas modernas possuem diversos mecanismos e recursos que resultam na construção de aplicações e códigos mais eficientes, porém também mais complexos. Estes recursos incluem suporte a máquinas virtuais, execução de código pela `BIOS` *e pelo chipset, enclaves isolados e até mesmo componentes de hardware externos. Neste capítulo, introduzimos estes mecanismos, os desafios que estes trazem para a monitoração de sistemas e também novas possibilidades para a construção de mecanismos e técnicas de monitoração e análise.*

3.1. Introdução

No mundo atual, os sistemas computacionais tem papel de destaque, visto que as capacidades destes suportam muitos dos aspectos da vida cotidiana, tal qual pode ser verificado pela adoção de sistemas de *Internet Banking* [G1 2017] e de iniciativas de governo eletrônico [do Brasil 2018]. No entanto, a medida em que cresce a dependência da sociedade com relação a estes sistemas, crescem também as ameças a estes e seus usuários. Os mesmos desenvolvimentos que trouxeram benefícios a sociedade, trouxeram novos riscos, como as fraudes bancárias digitais [Grégio et al. 2013] e o *cybercrime* [Kaspersky 2015]. Para lidar com estes novos desafios, organizações e usuários adotam soluções de análise [Sikorski and Honig 2012], que permitem a especialistas obter uma melhor compreensão das ameaças para o desenvolvimento de mecanismos de remediação, proteção e prevenção.

Contudo, a batalha entre atacantes e analistas constitui uma verdadeira corrida armamentista (*arms-race*), visto que os atacantes continuamente desenvolvem técnicas de anti-análise [Botacin et al. 2017] para evitar a descoberta e inspeção de seus artefatos. Deste modo, soluções de análise tradicionais [Botacin et al. 2018c] não são mais suficientes para uma cobertura completa das ameaças atuais, visto que o uso de técnicas de anti-análise tem crescido tanto no cenário nacional [Botacin et al. 2015] quanto global [Branco et al. 2012, Barbosa and Branco 2014]. Deste modo, o desenvolvimento de soluções de análise mais eficientes é fundamental para garantir a manutenção da segurança de aplicações atuais e futuras. Contudo, este desenvolvimento é desafiador, sobretudo para soluções completamente baseadas em *software*, dada a inerente possibilidade de subversão de um componente de *software* voltado a proteção por um componente de *software* malicioso.

Avanços recentes no desenvolvimento de *hardware* tem apresentado novos recursos atrelados aos processadores modernos, como máquinas virtuais de *hardware* e modos de execução isolados do restante do sistema. O surgimento destes recursos tem impulsionado o desenvolvimento de soluções de análise com suporte de *hardware* que podem proporcionar aos analistas recursos e capacidades que os levem um passo a frente dos atacantes. Desta forma, esse capitulo se dedica a apresentar estes recursos, discutir seus pontos fortes e limitações, bem como apontar oportunidades de pesquisa existentes.

Neste capítulo, consideramos procedimentos de análise em um sentido amplo, cobrindo desde mecanismos para engenharia reversa e *debugging*, em um sentido de análise mais estrito, até a monitoração de sistemas em tempo real, a detecção de ataques e procedimentos forenses, aplicações que apresentam requisitos de análise diversos.

Este capítulo foi planejado para ser apresentado de forma autocontida, isto é, não requerindo que o leitor consulte referências externas para sua compreensão. Contudo, assumimos que a audiência possui conhecimentos básicos na área de segurança e, em especial, na área de análise de aplicações, de modo que possamos partir do ponto das diferenças trazidas pelos desenvolvimentos recentes em relação aos já conhecidos pelos leitores.

De um modo geral, nossa contribuição é atualizar analistas, profissionais da área e estudantes em relação ao estado da arte em soluções de análise e apontar possibilidades

futuras que estes possam explorar em suas pesquisas e/ou atividades profissionais.

Este capítulo está organizado da seguinte forma: na Seção 3.2, apresentamos os principais recursos de inspeção e características das soluções de análise modernas. A implementação destes recursos é o objetivo do uso das tecnologias e técnicas apresentadas nas demais seções; na Seção 3.4, apresentamos diferentes mecanismos arquiteturais e tecnologias a serem usados para implementar os diferentes recursos de inspeção previamente apresentados; na Seção 3.3, apresentamos as técnicas de monitoração empregadas de modo a prover os recursos de inspeção mencionados através do uso das tecnologias anteriormente apresentadas; na Seção 3.5, discutimos aplicações de análise com base nas tecnologias e técnicas de implementação apresentadas; na Seção 3.6, traçamos um panorama das soluções apresentadas e discutimos problemas em aberto; finalmente, apresentamos nossas conclusões na Seção 3.7.

3.2. Recursos de Inspeção

Este capítulo versa sobre a análise de binários e sistemas. Contudo, o termo análise é suficientemente amplo para abranger distintos recursos de inspeção. Nesta seção, apresentamos os principais recursos presentes nas soluções modernas e seus respectivos cenários de aplicação.

3.2.1. Transparência

Soluções de análise que visem inspecionar aplicações específicas, tais como *debuggers* e analisadores de *malware*, devem ser transparentes ao objeto monitorado, isto é, não devem permitir que o objeto sob análise identifique sua execução em um ambiente monitorado. Isto é essencial para garantir a correta execução dos objetos, visto que muitas aplicações modernas são equipadas com técnicas de anti-análise, evitando, assim, procedimentos de engenharia reversa. As técnicas de anti-análise são empregadas tanto por aplicações legítimas, como *anti-cheats* de plataformas digitais [Li et al. 2004], quanto por aplicações maliciosas, como exemplares de *malware*, que visam evadir os processos de análise [Chen et al. 2008].

A identificação dos ambientes de análise se dá através de quatro principais mecanismos [Botacin et al. 2017]:

1. ***Fingerprinting***: Técnicas de *fingerprinting* consistem em localizar identificadores únicos no sistema alvo, como o IP público de um sistema de análise, permitindo o lançamento de ataques direcionados ou a evasão de procedimentos de análise. Este tipo de técnica é extensamente abordada na literatura [Liu et al. 2014] e não será coberta em profundidade neste capítulo.

2. **Verificação de integridade**: A verificação de integridade consiste em detectar a injeção de código no processo em execução—como nos procedimentos de injeção de bibliotecas (*DLL Injection*, por exemplo)—ou na alteração de estruturas do sistema—como no *hooking* de APIs. Desta forma, um exemplar de *malware* pode, por exemplo, detectar que uma biblioteca de monitoração foi carregada e, assim, terminar sua execução.

3. **Identificação de efeitos colaterais de execução em ambientes instrumentados**: A detecção de efeitos colaterais de execução consiste em identificar comportamentos anômalos do sistema devido a instrumentação deste para fins de monitoração. Muitas soluções de análise se baseiam em máquinas virtuais de *software* ou em emuladores, que não traduzem fielmente o comportamento dos sistemas reais. Por exemplo, instruções atômicas em máquinas reais podem ser executadas em múltiplos passos e, emuladores [Willems et al. 2012a], o que permite a um atacante agir maliciosamente apenas em máquinas reais e, portanto, atacar apenas os usuários finais. Tais instruções são conhecidas como "*red pills*" [Martignoni et al. 2009, Shi et al. 2014] e atualmente a identificação destas pode ser realizada de maneira automática [Paleari et al. 2009].

4. **Identificação de variações nas medidas de tempo**: A evasão por detecção de medidas divergentes de tempo se baseia no fato de que os códigos de instrumentação adicionados ao sistema—sejam no *kernel*, em bibliotecas ou no próprio emulador—consomem ciclos de CPU adicionais em relação a execução do código monitorado, resultando em variações no tempo de execução total do objeto monitorado. Um código que verifique seu próprio tempo de execução é capaz de inferir sua execução em um ambiente monitorado e, assim, evadir o processo de análise.

Formalmente, o conceito de transparência de um ambiente e da execução neste é definido por cinco requisitos [Dinaburg et al. 2008a]:

1. **Monitoração com privilégios superiores**: O mecanismo de análise deve ser mais privilegiado do que o objeto sendo monitorado. Como exemplo, a monitoração de recursos em nível de usuário deve ser implementada por mecanismos de *kernel*. Recursos de *kernel*, por sua vez, devem ser monitorados por uma entidade externa, como um *hypervisor*, e assim sucessivamente.

2. **Execução livre de efeitos colaterais não privilegiados**: Qualquer instrução que introduza um efeito colateral no sistema deve ser tratada por uma exceção capaz de esconder este efeito.

3. **Instruções com semânticas idênticas ao caso base**: Cada instrução executada no ambiente de monitoração deve ter o mesmo efeito e conduzir a execução da mesma instrução seguinte do que sua execução no ambiente não-monitorado.

4. **Tratamento transparente de exceções**: Dado uma exceção na *i-ésima* instrução, o fluxo de execução após o tratador deve retornar para a *(i+1)-ésima* instrução.

5. **Medidas de tempo idênticas**: A medida de tempo decorrido para a execução no ambiente monitorado e no ambiente não-monitorado devem ser idênticas.

Arquiteturas modernas proveem recursos de inspeção que permitem monitorar a execução de aplicações sem a necessidade de se injetar código, sendo, portanto, essenciais para o desenvolvimento de soluções transparentes. Além disto, recursos presentes nas arquiteturas modernas, como máquinas virtuais de *hardware*, permitem que o código

monitorado seja executado no processador real, não apresentando, assim, efeitos colaterais de execução. O uso destes recursos para fins de análise transparente é descrito em detalhes nas seções a seguir.

3.2.2. Amplitude de monitoração

Cada modelo de ameaça apresenta diferentes requisitos quanto a amplitude da monitoração, isto é, das diferentes fontes de dados requeridas para a monitoração. Por exemplo, enquanto a detecção de comportamento anômalo pode ser implementada a nível de aplicação, procedimentos forenses podem requerir o *dump* da memória do *kernel*. A implementação de um mecanismo de análise em cada um dos níveis requer a compreensão das possibilidades e limitações arquiteturais das plataformas modernas.

Processadores modernos isolam os diferentes níveis de execução em *rings*. As aplicações de nível de usuário são executadas em `ring` 3, ao passo em que o *kernel* é executado no `ring` 0. Os *rings* 1 e 2 são dedicados a *drivers* e bibliotecas, mas não são utilizados, na prática, pelos sistemas modernos. Os *rings* são ilustrados pela Figura 3.1.

Os privilégios de execução de um nível permitem que este dê garantias sobre os demais. Por exemplo, o *kernel* é capaz de monitorar o nível de usuário sem que este possa subverter o *kernel*, por estar em um nível de execução inferior e, portanto, isolado por *hardware*. O isolamento entre os níveis se dá, por exemplo, através do impedimento do mapeamento de uma região de memória pertencente a uma aplicação operando em um *ring* superior ao que a aplicação solicitante opera.

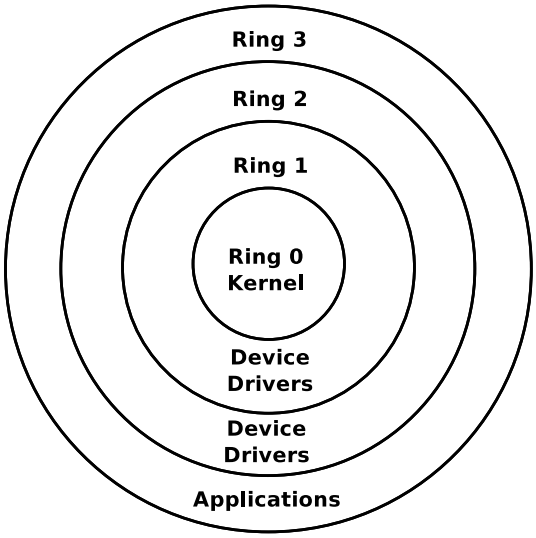

Figura 3.1. Privilégios de Execução. O *ring* 0 é o mais privilegiado e pode monitorar os demais *rings*. As aplicações executam em *ring* 3 e não podem interferir com os demais *rings*.

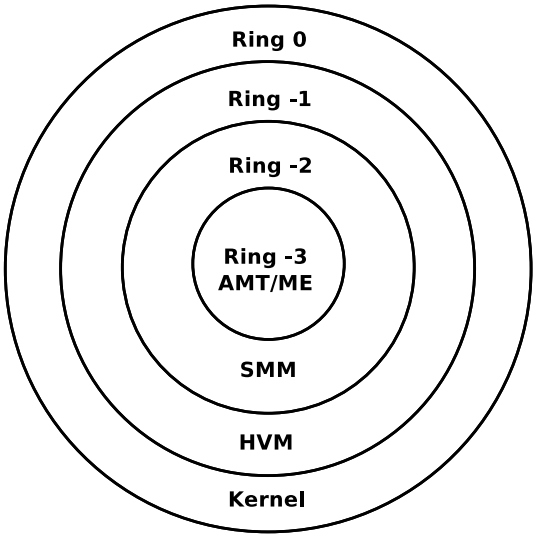

Figura 3.2. Novos *rings* privilegiados. O *ring* -3 é o mais privilegiado e pode monitorar os outros *rings*. Nesta nova configuração, o *kernel* executa dentro de uma máquina virtual de *hardware* (HVM), em *ring* -1.

Sistemas de análise modernos apresentam modelos de ameaças cada vez mais amplos e, com esta necessidade, novos *rings* vem sendo propostos. O uso de máquinas

virtuais de *hardware*—*Hardware Virtual Machines* (HVM)—possibilita que um *hypervisor* monitore tanto o *kernel* quanto o modo usuário do sistema *guest*. Portanto, sistemas baseados em HVM vem sendo chamados de *ring* -1. Similarmente, sistemas baseados na instrumentação do modo SMM—*System Management Mode*—, presente na BIOS, são capazes de monitorar tanto o *kernel* e modo usuário quanto o próprio *hypervisor* em execução no sistema. Desta forma, o modo SMM é denominado *ring* -2. Finalmente, o modo ME (*Management Engine*) dos *chipsets* é capaz de monitorar inclusive o funcionamento da BIOS, sendo, portanto, denominado *ring* -3. Estes novos *rings* são ilustrados pela Figura 3.2.

A monitoração entre os diferentes níveis, contudo, impõe um desafio significativo: a interpretação dos dados, já que que cada nível apresenta um `nível de abstração` diferente. Por exemplo, enquanto a monitoração em nível de usuário é capaz de identificar um dado evento como sendo uma chamada de função, a monitoração do mesmo evento em nível de HVM é capaz apenas de identificar a execução de um dado endereço. As Figuras 3.3 e 3.4 ilustram, respectivamente, as tecnologias de análise apresentadas neste capítulo e seus níveis de abstração.

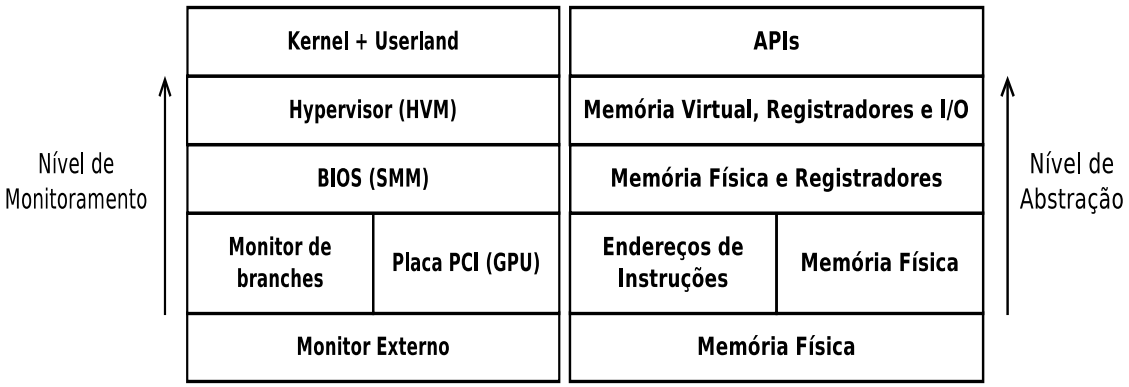

Figura 3.3. Níveis de monitoramento. Cada tecnologia apresentada neste capítulo monitora o sistema em um nível diferente. Os níveis mais altos se aproximam de soluções de *software*, enquanto níveis inferiores operam mais próximo do *hardware*.

Figura 3.4. Níveis de abstração. Cada tecnologia apresentada neste capítulo opera em um nível de abstração diferente. Quanto mais alto o nível, mais próximo de uma informação compreensível para o ser humano. A passagem da informação de um nível de abstração para o outro exige procedimentos de introspecção para a adequação da sua representação.

A diferença de representações entre os níveis, chamada de *gap* semântico, requer que procedimentos de "enriquecimento" dos dados sejam desenvolvidos para aumentar o nível de abstração das informações coletadas. Por exemplo, um sistema HVM precisa previamente mapear endereços em chamadas de função de modo a identificar as chamadas quando monitorando o sistema em tempo real através dos endereços executados. Este procedimento de "enriquecimento" das informações coletadas em um nível de abstração inferior para interpretação em um nível de abstração superior é chamado introspecção.

Ferramentas de análise podem se basear em soluções conhecidas de introspecção, como a LibVMI [LibVMI 2015], ou desenvolver suas próprias soluções [Botacin et al. 2018a].

De uma forma geral, mecanismos de introspecção podem ser classificados em quatro categorias [More and Tapaswi 2014]:

1. **Memória**: A introspecção de memória consiste em reconstruir estruturas a partir das visões de memória obtidas em cada nível. Este tipo de técnica permite, por exemplo, interpretar séries de apontadores como elementos de uma lista de processos. Este tipo de técnica é frequentemente usado por soluções forenses para a reconstrução de contexto a partir de *dumps* de memória.

2. **I/O**: De forma similar a introspecção de processos, procedimentos de reconstrução podem ser aplicados a eventos de I/O para reconstruir a estrutura dos dados sendo transferidos. Este tipo de técnica é frequentemente utilizado por soluções que imponham políticas de segurança durante a transmissão de dados.

3. **Chamadas de sistema**: A introspecção a nível de chamadas de sistema (*syscalls*) é focada em reconstrução de contextos, tais como os processos que invocam uma dada *system call* e seus respectivos argumentos. Dado que processos são uma abstração de alto nível, a identificação destes ocorre através da associação destes com valores em *hardware*, tais como o valor do registrador `CR3`, usado como apontador da tabela de páginas de memória e, portanto, único para cada processo. Este tipo de técnica é frequentemente utilizado por soluções de monitoramento HVM e SMM.

4. **Processos**: A introspecção a nível de processo objetiva obter informações específicas de cada processo, como os recursos acessados por cada processo analisado. Este tipo de introspecção é frequentemente utilizado por soluções antivírus para a detecção de *malware* em nível de espaço de usuário (*userland*).

Na prática, a literatura acadêmica apresenta diversas soluções de análise baseadas em introspecção, tais como Anubis [Bayer et al. 2006, ISECLAB 2010] e Danubis [Neugschwandtner et al. 2010], implementados sobre QEMU [Bellard 2005], e Bitblaze [Song et al. 2008] e Virtice [Quynh and Suzaki 2010], sobre TEMU. Contudo, mecanismos de introspecção se tornaram fundamentais a partir da popularização do uso dos novos recursos de *hardware* para a implementação dos mecanismos de análise. O uso de introspecção pelas diversas soluções são detalhados nas seções a seguir.

3.2.3. Posicionamento e capacidade de ação

Sistemas computacionais modernos são organizados em diversas camadas e o monitoramento em cada uma delas apresenta vantagens e desvantagens significativas. Mais do que abstrações, essas camadas, de fato, cumprem diferentes funções e, portanto, apresentam diferentes desafios para suas instrumentações e impõem diferentes custos. O monitoramento em alto nível, tal como a nível de sistema operacional, apresenta baixo custo de implementação, já que a solução de análise pode utilizar bibliotecas de sistema. Por outro lado, este tipo de monitoramento é mais suscetível a subversão por mecanismos mais privilegiados. O monitoramento em baixo nível, por sua vez, apresenta maior proteção contra

subversão, porém, seus custos de implementação são maiores, dado a limitada possibilidade de reuso de código e os conhecimentos de implementação exigidos. Desta forma, encontrar um balanço entre o nível de proteção requerido por um dado modelo de ameaça e o custo de sua implementação é um desafio significativo.

Além disso, a camada em que o sistema de análise é posicionado influencia a capacidade de ação deste. Sistemas posicionados no mesmo nível do sistema monitorado são capazes de bloquear a execução caso uma ameaça seja detectada. Sistemas posicionados externamente ao objeto monitorado podem ser incapazes de bloquear a execução de um objeto interno, embora possuam capacidade de análise do mesmo. Este cenário é observado quando se delega a tarefa de monitoração a uma entidade de *hardware* externa ao mecanismo de análise, tal como um co-processador. Uma discussão aprofundada deste tipo de solução é apresentada nas seções a seguir.

3.2.4. Carregamento em tempo real

Diferentes tarefas de análise apresentam diferentes requisitos para as ferramentas a serem utilizadas. Enquanto tarefas como análise de *malware* são realizadas em ambientes controlados e previamente configurados, tarefas como a forense são frequentemente realizadas em campo, apresentando significativas restrições para a instalação e configuração de ferramentas no ambiente. Desta forma, uma característica desejável para soluções de análise, em especial as voltadas à forense computacional, é o carregamento da solução em tempo real. Também chamado de *live-loading*, este tipo de carregamento permite ao perito forense inserir a solução de análise no sistema alvo e utilizar o próprio sistema para a inspeção, com garantias de que o carregamento da solução não interfira nos artefatos a serem coletados. Este tipo de carregamento é possível, por exemplo, através do uso de máquinas virtuais de *hardware*, que permitem transformar o sistema *host* em um *guest* da mesma máquina. A implementação deste tipo de recurso é apresentada nas seções a seguir.

3.2.5. Base de código confiável

Soluções de análise requerem um determinado conjunto de códigos para seu funcionamento. Este conjunto inclui todos os códigos que suportam sua operação, desde os códigos responsáveis por implementar a tarefa de monitoração em si aos *frameworks* sob os quais a solução é construída. Este conjunto é denominado base de código confiável (*Trusted Code Base*—TCB), uma vez que deve ser considerado como legítimo e íntegro *a priori*. Quanto mais se confia em recursos de terceiros, maior o TCB. Por exemplo, um *driver* de monitoração operando em *kernel* deve considerar todo o *kernel* como seu TCB, visto que subversões deste podem acarretar em subversões do mecanismo de análise em si.

De um modo geral, soluções de análise visam reduzir o TCB, ampliando suas garantias de segurança e também seu modelo de ameaças. Contudo, estas devem buscar um balanço entre a diminuição do TCB e do aumento dos custos de implementação, visto que, ao não se utilizar de códigos terceiros, a solução deve implementar seus próprios códigos para todas as tarefas. Uma solução em BIOS, por exemplo, enquanto apresenta um TCB minimal, requer a implementação até mesmo de protocolos de comunicação,

visto que este suporte não é provido pela BIOS do sistema.

A Figura 3.5 ilustra o tamanho da base de código confiável (em número de linhas de código) para algumas das soluções apresentadas neste capítulo. Uma solução operando a nível de *kernel* (Linux) requer a inclusão de mais de 10 milhões de linhas de código em seu TCB para contar com os recursos e facilidades oferecidos por este. Uma solução operando a nível de máquina virtual (Xen) requer a inclusão de mais de 500 mil linhas para fazer uso de uma solução de virtualização completa, com todos os *drivers* de dispositivos. Como alternativa para reduzir o TCB, soluções de propósito específico podem ser desenvolvidas (MAVMM [Nguyen et al. 2009]). Neste caso, recursos opcionais e *drivers* não utilizados são removidos da base de código. Soluções inteiramente baseadas em hardware externo (Ki-mon [Lee et al. 2013]) requerem menos de 1000 linhas de código, visto que estas não se baseiam em nenhum suporte *a priori*.

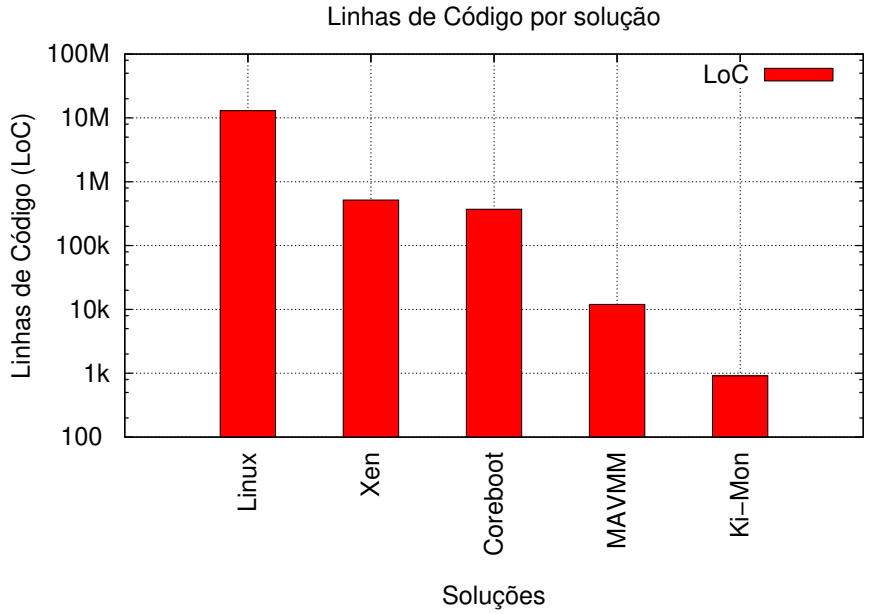

Figura 3.5. Base de código confiável (em número de linhas de código) por aplicação. Quanto maior o nível de abstração, maior a base de código responsável por implementar a solução de monitoramento, portanto, maior a base de código que deve ser confiada (TCB). Como exceção à regra, ferramentas podem ser desenvolvidas com o propósito de minimizar o TCB, como no caso da MAVMM.

3.2.6. Manutenção de integridade e operação

Um sistema voltado a prover garantias de segurança deve manter sua integridade. Uma solução de segurança comprometida pode ser incapaz de detectar ataques, prover informações imprecisas ao analista ou mesmo invalidar perícias forenses. A manutenção de integridade, contudo, é desafiadora em múltiplos aspectos: i) quanto a manutenção do desempenho, uma vez que a solução deve gastar ciclos de CPU para computar *hashes* que atestem sua própria integridade; ii) quanto a operação em meios não confiáveis, uma vez que as soluções atuam diretamente com componentes comprometidos. Uma solução que envie os dados coletados em um sistema para um agente externo deve criptografar sua comunicação uma vez que a interface de rede pode ser comprometida pelo atacante, o

que afetaria os pacotes trafegados. A operação em meios não confiáveis é particularmente desafiadora quando a solução é carregada em um ambiente previamente comprometidos, como em perícias forenses. Nestes casos, a solução deve validar as informações coletadas através da correlação das informações obtidas através de múltiplos fontes de dados, em um processo conhecido como detector de mentiras (*lie detector*). Por exemplo, uma solução baseada em máquinas virtuais pode coletar informações interna e externamente (por introspecção) ao sistema monitorado e considerá-las válidas se estas coincidirem. Este processo atesta que a informação provida é legítima, uma vez que o *hypervisor* faz parte do TCB e, portanto, não foi comprometido.

3.2.7. Sincronia

Uma decisão de projeto importante para se implementar uma solução de monitoração é o tipo de mecanismo de captura e coleta de dados a ser utilizado: síncronos ou assíncronos. Mecanismos síncronos coletam dados em intervalos determinados, sendo, geralmente, implementados através de *polling*. Desta forma, estes são adequados para tarefas que podem ser delegadas a etapas de processamento posteriores (pós-processamento), como o "enriquecimento" dos dados coletados em um traço de execução. Mecanismos assíncronos, por sua vez, coletam os dados quanto determinados eventos ocorrem (*event-driven*), sendo, portanto, adequados para soluções em tempo real, como *debuggers* ou detectores de ataques. Soluções síncronas não são adequadas para mecanismos de tempo real pois um ataque poderia ocorrer no intervalo entre 2 inspeções [Moon et al. 2012], como exemplificado na Figura 3.6.

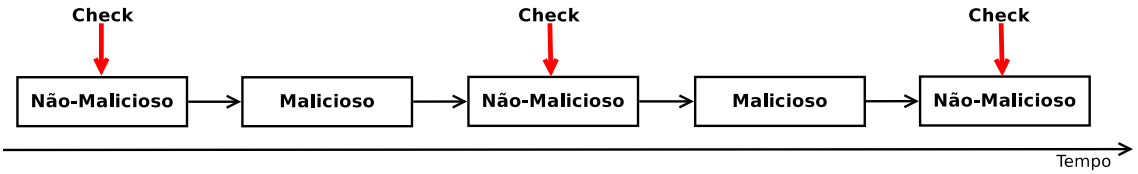

Figura 3.6. Evasão de soluções do tipo *snapshot*. Em soluções que implementem verificações síncronas, ações maliciosas podem não ser detectadas caso estas ocorram no intervalo entre duas inspeções.

Soluções síncronas, no entanto, ainda são populares pois a implementação de mecanismos assíncronos é desafiadora, uma vez que eventos precisos devem ser escolhidos como gatilhos de inspeção. Arquiteturas modernas oferecem suporte em *hardware* para a implementação deste tipo de gatilho através do lançamento de interrupções programáveis. Estas são descritas em detalhes nas seções a seguir.

Desempenho. Enquanto o foco principal das ferramentas de análise são os aspectos de segurança, requisitos de desempenho não podem ser desprezados para muitas aplicações, notavelmente as de operação em tempo-real. Soluções que busquem aplicar políticas de segurança ou detectar ataques em tempo real não devem consumir um tempo significativo processando suas rotinas para não degradar o desempenho da aplicação sendo monitorada. Arquiteturas modernas possuem recursos como contadores de *performance* em hardware que podem ser utilizados para capturar dados com baixo impacto no desempenho, podendo, portanto, serem utilizados para a implementação de mecanismos de análise em tempo real [Opsahl 2013]. Maiores detalhes da aplicação destes são apresentados na

seções a seguir.

3.2.8. Dispensa de atualizações frequentes

Uma característica desejável para os sistemas de análise é que, uma vez desenvolvidos, estes não exijam reestruturação e/ou recompilações frequentes. Este tipo de limitação é presente em abordagens de detecção baseadas em assinatura.

Frequentemente, as soluções de análise e monitoração de artefatos maliciosos [Vasudevan and Yerraballi 2006b, Vasudevan and Yerraballi 2006a] propõem mitigar os efeitos colaterais de execução e de técnicas de anti-análise através de instrumentação dinâmica , utilizando-se, por exemplo, das soluções DynamoRIO [DynamoRIO 2001] e PIN [Intel 2015]. Apesar de eficazes contra as técnicas conhecidas, o processo de mitigação não é escalável [Kang et al. 2009], visto que a descoberta de novas técnicas de anti-análise requerem que novas mitigações sejam implementadas na solução. Arquiteturas modernas tornam as soluções de análises escaláveis, dado que estas não precisam implementar contramedidas para efeitos colaterais de execução, pois estas ocorrem em ambientes de processamento nativos.

3.2.9. Compatibilidade e Integração

Apesar das limitações impostas pelo TCB em relação ao sistema de análise, componentes externos ao sistema são frequentemente construídos sobre outras soluções, reduzindo, portanto, os custos de implementação e propiciando suporte a sistemas e padrões legados. De um modo geral, as soluções de análise atuais [Zhang et al. 2015, Wang et al. 2011] são integradas ao *frontend* GDB, proporcionando que este manipule os elementos arquiteturais da plataforma em *backend*.

3.3. Tecnologias e Implementações

Nesta seção, apresentamos tecnologias e recursos arquiteturais presentes em plataformas modernas que podem ser utilizados para fins de análise.

3.3.1. Máquinas Virtuais de *Hardware*

Hardware Virtual Machines (HVM) são extensões dos modos de operação presentes nos processadores modernos. Tipicamente, processadores `x86-64` apresentam três modos de operação:

1. **Modo de endereçamento real**: Modo de operação original da arquitetura `x86`. Este modo é ativado quando o computador é inicializado (*boot*) e não conta com proteções de memória.

2. **Modo protegido**: Extensão do modo original da arquitetura `x86` e modo nativo de operação dos processadores modernos, fornecendo suporte a memória virtual e paginação.

3. **Modo SMM**: Modo de gerência de recursos do processador, como controle de energia e de temperatura. O código deste modo é executado pela BIOS e pode ser instrumentado, como mostrado nas seções a seguir.

As instruções de virtualização implementadas pelas tecnologias Intel VT-x [Intel 2013] e AMD-v/SVM [AMD 2013] adicionam ao processador dois novos modos de operação: *VM-root* e *VM-non-root*. Estes modos correspondem a operação do *hypervisor* e do *guest*, respectivamente, como mostrado na Figura 3.7, e a transição entre estes modos é dada por *VM-EXITs* (*non-root* para *root*) e *VM-ENTRY* (*non-root*) para *root*), como mostrado na Figura 3.8. As ações que resultam em saídas do *guest* para o *hypervisor* são configuráveis pelo *hypervisor* e tratadas por este através de rotinas instrumentáveis. Desta forma, o *hypervisor* pode monitorar ações específicas da máquina virtual, tais como escritas em registradores e instruções executadas.

Ao contrário de máquinas virtuais de *software*, o *hypervisor* conta com assistência do processador para a identificação e tratamento dos eventos, resultando em um significativo ganho de desempenho. Por exemplo, o próprio *hardware* é responsável por realizar a cópia dos registradores em cada troca de contexto. Além disso, as instruções de ambos os modos (*root* e *non-root*) são executados pelo processador nativo, sem tradução ou emulação, o que torna a execução, ainda que monitorada pelo *hypervisor*, livre de efeitos colaterais.

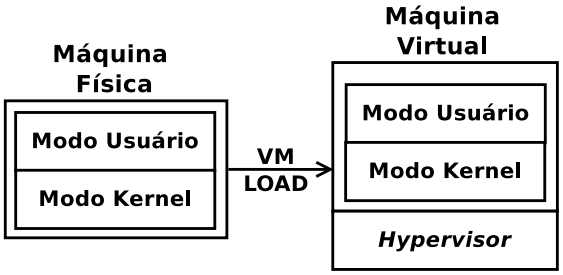

Figura 3.7. Migração do sistema para máquinas virtuais. Quando operando em uma máquina virtual, tanto o modo *kernel* quando o modo usuário ficam sobre supervisão de um *hypervisor*.

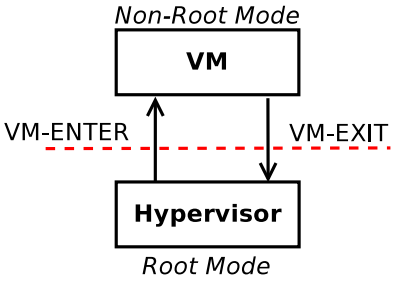

Figura 3.8. Novos modos de operação. A instrução `vmload` inicia a operação da máquina virtual. Quando uma ação monitorada ocorre no sistema *guest*, uma saída para o *hypervisor* é causada.

HVMs apresentam significativa diferença em relação a VMs tradicionais quanto ao gerenciamento de memória. Enquanto em VMs tradicionais a memória física do sistema *guest* é diretamente traduzida para memória física do *host*, HVMs apresentam um mecanismo de tradução dupla, que resulta em uma mapeamento da memória física do *guest* em memória virtual do *host*, como mostrado na Figura 3.9. O mecanismo de tradução implementado pelo processador pode também ser instrumentado, possibilitando o desenvolvimento de mecanismos de análise de memória. Complementarmente, HVMs virtualizam em *hardware* as funções de I/O através da IOMMU (*Input Output Memory Management Unity*), o que permite monitorar, também, os demais acessos do sistema, tais como acessos diretos à memória (DMA).

O carregamento de HVMs se dá através de *drivers* de *kernel* que configuram os registradores em *hardware* para a operação no modo de máquina virtual. Esta característica permite que sistemas em execução sejam transformados em VMs em tempo de execução (*live loading*), bastando ao *hardware* copiar todos os registradores da máquina

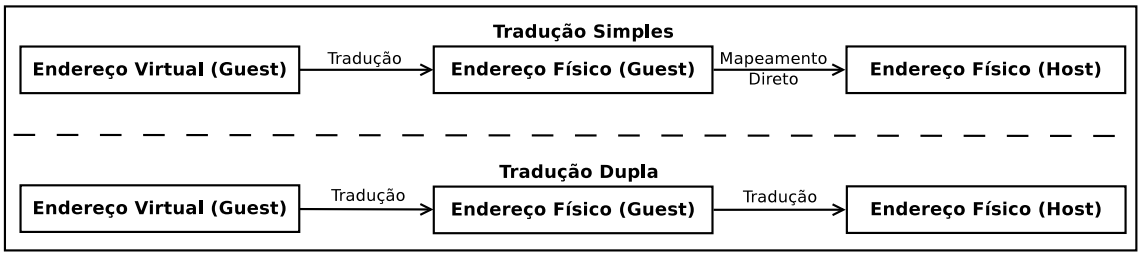

Figura 3.9. Mecanismos de tradução de memória de um HVM. Enquanto máquinas virtuais por *software* apresentam apenas uma etapa de tradução, no *guest*, HVMs apresentam duas etapas, incluindo uma etapa adicional no *hypervisor*. Este mecanismo adicional de tradução pode ser instrumentado de modo a permitir o monitoramento de memória.

física para o contexto da máquina virtual. A Figura 3.10 ilustra o bloco de controle (*Virtual Machine Control Structure*—VMCS) das máquinas virtuais tal qual definido pela Intel [Intel 2013]. Para fins de monitoração, este bloco de controle deve ser preenchido com as ações que causem saídas do *hypervisor*. O código 3.1 ilustra este preenchimento pela solução de virtualização `Xen` [Project 2015].

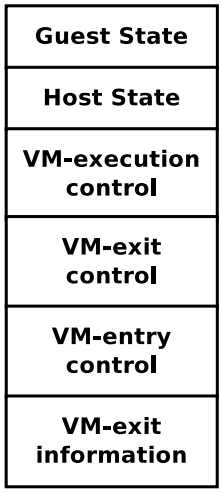

Figura 3.10. Estrutura de Controle da máquina virtual. Cada bloco da estrutura armazena informações relativas a diferentes aspectos da operação. Por exemplo, o *guest state* armazena informações dos registradores enquanto o *exit control* define os eventos que serão monitorados.

```
1  int vmx_init_vmcs_config(){
2  min = (CPU_BASED_HLT_EXITING |
3  CPU_BASED_VIRTUAL_INTR_PENDING |
4  CPU_BASED_CR8_LOAD_EXITING |
5  CPU_BASED_CR8_STORE_EXITING |
6  CPU_BASED_INVLPG_EXITING |
7  CPU_BASED_CR3_LOAD_EXITING |
8  CPU_BASED_CR3_STORE_EXITING |
9  CPU_BASED_MONITOR_EXITING |
10 CPU_BASED_MWAIT_EXITING |
11 CPU_BASED_MOV_DR_EXITING |
12 CPU_BASED_ACTIVATE_IO_BITMAP |
13 CPU_BASED_USE_TSC_OFFSETING |
14 CPU_BASED_RDTSC_EXITING);
```

Código 3.1. Código do Xen. *Virtual Machine Control Structure* (VMCS) deve ser configurado com os eventos que causaram saídas do *hypervisor* e, portanto, permitirão a monitoração.

3.3.2. Modo SMM

O modo de gerência da BIOS (*System Management Mode*) é um modo de operação do processador voltado a tarefas de gerência, como controle de energia e de temperatura. O código que opera em modo SMM é residente na BIOS, não sendo acessível a partir de outros modos de operação do processador. Entradas no modo SMM são possíveis apenas

através de interrupções especiais do modo SMM denominadas SMI. SMIs podem ser geradas através de escritas em pinos especiais do processador, por dispositivos PCI ou pela redireção de interrupções do *chipset* para a BIOS. A Figura 3.11 ilustra a possibilidade de se configurar as interrupções dos contadores de *performance* para serem entregues como SMIs. Neste caso, eles seriam tratadas por um tratador de interrupções tal qual mostrado no Código 3.2.

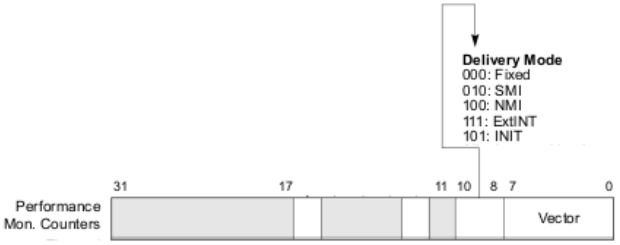

Figura 3.11. Geração de SMIs. Interrupções de diversos tipos, como as geradas pelos contadores de *performance*, podem ser entregues via SMIs e tratadas pelo modo SMM da BIOS.

```
1 void smi_handler(u32
      smm_revision) {
2 unsigned int node;
3 smm_state_save_area_t
      state_save;
4 node=nodeid();
```

Código 3.2. Código do Coreboot. Interrupções do tipo SMI tem um tratador especial dentro da BIOS. Este pode ser instrumentado para obter a causa do evento (node) e realizar tarefas de monitoração e imposição de políticas de segurança.

Como o modo SMM não é endereçado pelos demais modos de operação, este é protegido de subversão, tornando-se atraente para a implementação de soluções de análise de artefatos maliciosos. Além disso, tal qual HVMs, o modo SMM permite a execução de código no processador nativo, sendo livre de efeitos colaterais de execução. Como distinção do modo HVM, que apresenta tradução dupla de endereços em *hardware*, o modo SMM é capaz apenas de endereçar memória física. Além disso, as interrupções SMIs não entregam diretamente seu motivo causador, requerindo, portanto, procedimentos de introspecção mais complexos para a identificação dos eventos. A operação no modo SMM também é mais exigente quanto a implementação, já que a BIOS não fornece qualquer suporte prévio para extensões. Desta forma, recursos típicos de soluções de análise, como protocolos de comunicação e funções de *hash* devem ser inteiramente implementados pela solução de análise. As soluções mais populares para a reescrita de BIOS são Coreboot [CoreBoot 2015] e SeaBios [SeaBIOS 2015].

3.3.3. Modo ME

O modo ME (*Management Engine*) é um dos modos de gerenciamento de sistema presente nas arquiteturas Intel, sendo, também, responsável por controles de temperatura e energia. Ao contrário dos modos HVM e SMM, que são implementados pela CPU, o modo ME é implementado pelo *chipset*, podendo, portanto, controlar o processador como um todo, independente do modo de operação deste (*kernel*, HVM ou SMM). O modo ME opera autonomamente e conta com suas próprias unidades de processamento (ALU), memória (RAM/ROM), *timer* e DMA. Similar ao modo ME, plataformas AMD apresentam um modo de controle denominado *Secure Processor* [AMD 2016].

3.3.4. Contadores de *performance*

Hardware Performance Counters (HPCs) são recursos de *hardware* internos ao processador que registram a ocorrência de eventos no sistema, desde instruções executadas a acessos de memória. A operação dos contadores ocorre em paralelo a execução de instruções no processador e, por isso, esta não impacta significativamente o desempenho do sistema. Por isso, os contadores de *performance* são frequentemente utilizados em soluções de análise em tempo real. As soluções baseadas em contadores de *performance* podem ser classificados em dois modos de operação: i) por amostragem; e ii) por evento.

Contadores com operação por amostragem, tais como o Intel PEBS (*Precise Event Based Sampling*), contam a quantidade de ocorrências de um dado evento em um determinado período de tempo. A coleta sucessiva destes dados em uma série temporal permite construir um perfil da operação normal do sistema e depois compará-lo com dados obtidos em tempo real, detectando, assim, comportamentos anômalos.

Contadores com operação orientada a eventos, tais como os monitores de *branch* Intel *Last Branch Record* (LBR) e *Branch Trace Store* (BTS) e o monitor de sistema *Processor Tracer* (PT), registram a ocorrência de eventos individuais e informações sobre estes. Os monitores de *branch*, por exemplo, são capazes de registrar endereços de origem e destino das instruções de desvio de fluxo. Estes endereços podem ser utilizados, por exemplo, para verificar a integridade do fluxo de controle da execução.

Independente do modo de coleta (por amostragem ou por evento), os contadores operam de forma similar. Os eventos são armazenados em registradores especiais do processador (*Model Specific Registers*—MSRs) ou em páginas de memória do sistema operacional e uma interrupção é gerada quando um *threshold* de armazenamento pré-estabelecido (em número de eventos) é atingido, como mostrado na Figura 3.12. Destaca-se que a coleta de dados pelos contadores ocorre de maneira global no sistema, não havendo separação dos dados por processos, requerindo, portanto, procedimentos específicos e de introspecção para esta tarefa [Botacin et al. 2018a]. Apesar disto, os contadores de *performance* permitem filtrar os tipos de eventos monitorados e o nível de ocorrência, como mostrado na Figura 3.13. Este tipo de filtragem permite, por exemplo, monitorar apenas aplicações em modo usuário ou apenas o *kernel* do sistema.

3.3.5. Enclaves Isolados

A possibilidade trazida pelos modos HVM, SMM e ME/AMT de se monitorar o sistema como um todo exigiu o desenvolvimento de ambientes específicos para a execução de aplicações que requerem fortes garantias de segurança, como a gerência de chaves criptográficas, visto que estes modos poderiam ser utilizados por atacantes para subverter a operação deste tipo de aplicação e assim, por exemplo, vazar chaves criptográficas e segredos, comprometendo a segurança do sistema.

Considerando este cenário, enclaves isolados foram propostos, tais como o Intel *Software Guard Extensions* (SGX) [Aumasson and Merino 2016, Mandt et al. 2016] e o ARM TrustZone [ARM 2009]. Nestes ambientes, as páginas de memória são isoladas pelo controlador de memória em *hardware* (*Memory Management Unity*—MMU), criptografadas durante as trocas de contexto e destruídas após sua utilização, de modo que

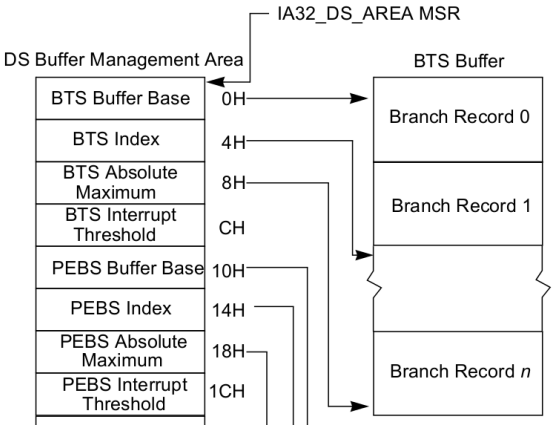

Figura 3.12. Configuração do monitor de *branch*. No monitor BTS, os dados são armazenados em páginas de memória. Quando o armazenamento atinge um *threshold* pré-estabelecido, uma interrupção é gerada. Fonte: Manual da Intel [Intel 2013].

Bit Field	Bit Offset	Access	Description
CPL_EQ_0	0	R/W	When set, do not capture branches occurring in ring 0
CPL_NEQ_0	1	R/W	When set, do not capture branches occurring in ring >0
JCC	2	R/W	When set, do not capture conditional branches
NEAR_REL_CALL	3	R/W	When set, do not capture near relative calls
NEAR_IND_CALL	4	R/W	When set, do not capture near indirect calls
NEAR_RET	5	R/W	When set, do not capture near returns
NEAR_IND_JMP	6	R/W	When set, do not capture near indirect jumps
NEAR_REL_JMP	7	R/W	When set, do not capture near relative jumps
FAR_BRANCH	8	R/W	When set, do not capture far branches
Reserved	63:9		Must be zero

Figura 3.13. Filtragem de eventos dos contadores de *performance*. Os contadores de *performance*, dentre os quais os monitores de *branch*, podem filtrar os eventos monitorados por tipo (*branches* diretos, indiretos, chamadas de função) ou por nível de ocorrência (*kernel* ou modo usuário). Fonte: Manual da Intel [Intel 2013].

outras aplicações não possam interferir com a aplicação em execução dentro do enclave, tal qual ilustrado pela Figura 3.14.

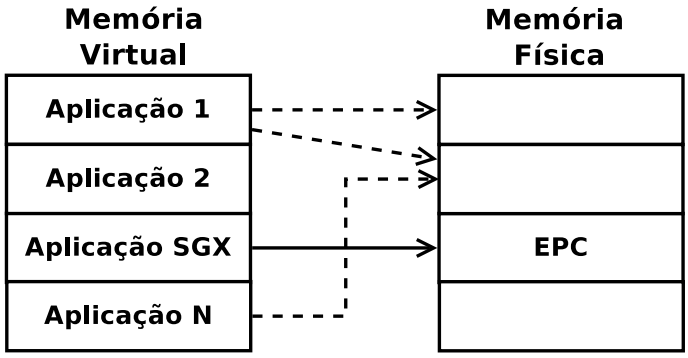

Figura 3.14. Proteção de memória dos enclaves SGX. O gerenciamento de memória dos enclaves (MMU e TLB) impede que outras aplicações mapeiem, direta ou indiretamente, a memória alocada para o enclave (*Enclave Page Cache*—EPC).

Para mover aplicações para dentro de um enclave SGX isolado, as aplicações precisam ser recompiladas de modo a suportar as instruções SGX, o que impede seu uso em aplicações legadas. Este capítulo visa discutir os impactos de segurança trazidos pela introdução dos enclaves isolados. Para informações sobre a programação neste ambiente, consulte [Will et al. 2017].

3.3.6. Recursos de *Hardware*

Além de modos de operação específicos, recursos individuais de *hardware* originalmente projetados para outros propósitos podem ser utilizados para auxiliar mecanismos de análise. Placas de expansão PCI, de um modo geral, e, em particular, GPUs (*Graphic Processing Unity*), possuem acesso direto a memória (*Direct Memory Access*—DMA), o que lhes permite mapear a memória do sistema como um todo, tanto para leitura quanto para escrita, como mostrado na Figura 3.15.

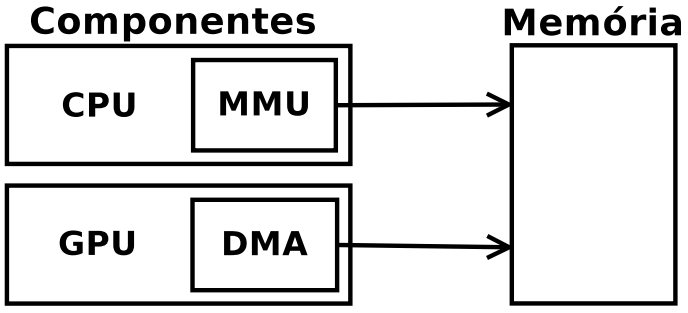

Figura 3.15. Acesso direto à memória. Este recurso permite que placas PCI, tais como GPU, acessem a memória diretamente. Como o acesso não é protegido pela MMU ou outro componente, os dispositivos PCI podem mapear a memória do sistema como um todo, incluindo as mesmas regiões acessadas pela CPU. A enumeração da memória como um todo permite, por exemplo, o *dump* para fins de forense.

O acesso direto a memória foi originalmente projetado para permitir a leitura e es-

crita dos *buffers* das placas de modo assíncrono, sem penalizar o desempenho do sistema. No entanto, por ser capaz de enumerar a memória de forma completa e sem restrições por parte da MMU ou outro componente, este recurso pode, também, ser utilizado, por exemplo, para fazer um *dump* completo da memória do sistema, um recurso desejável para muitos procedimentos forenses.

3.3.7. *Hardware* Externo

Em cenários onde o suporte arquitetural é limitado, o uso de *hardware* externo é uma alternativa para a implementação de mecanismos de segurança. Tipicamente, este tipo de abordagem é implementada por co-processadores responsáveis por realizar o monitoramento (*snooping*) do tráfego no barramento de memória compartilhado entre o processador monitor e o processador monitorado, como mostrado na Figura 3.16.

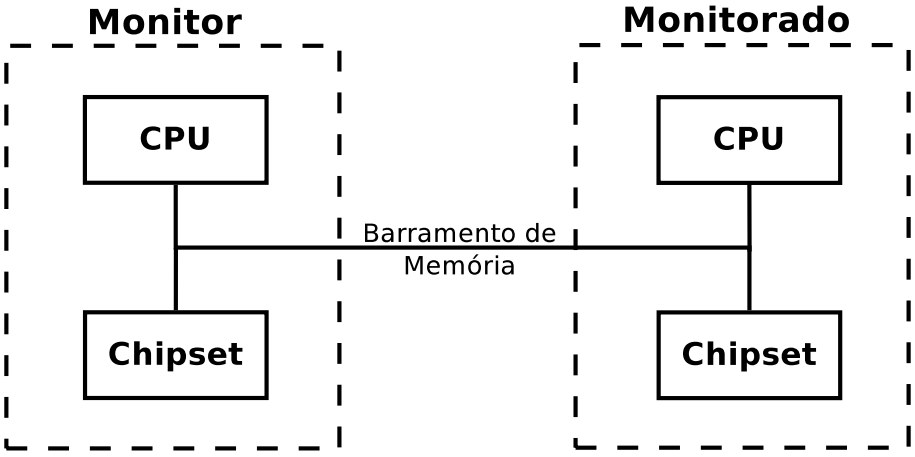

Figura 3.16. Monitoração com *hardware* externo. Dispositivos externos podem ser conectados a um barramento de memória compartilhado com a plataforma monitorada para realizar *snooping* dos dados.

Por estar posicionado externamente ao sistema monitorado, este tipo de solução não dispõe de informações de registradores, de modo que a reconstrução de contexto deve ser feita por procedimentos de introspecção a partir apenas dos dados em memória, uma tarefa significativamente mais complexa.

3.4. Técnicas de Monitoração

Embora as arquiteturas modernas apresentem diversos recursos que podem suportar procedimentos de análise, a efetiva aplicação destes pelas soluções requer o uso de técnicas de monitoração específicas. Nesta seção, apresentamos as técnicas mais frequentemente utilizadas.

3.4.1. Monitoração de eventos

Como as soluções de análise modernas são orientadas a eventos, a implementação da notificação da ocorrência dos eventos é uma etapa de desenvolvimento fundamental. Neste sentido, soluções HVM proveem um modo nativo de identificar a ocorrência de eventos, dado que cada `VM-EXIT` especifica a causa de sua ocorrência, como mudanças nos

valores do registrador `CR3`, interrupções geradas, escritas em regiões de memória monitoradas, entre outras. O modo SMM, por sua vez, não possui mecanismos nativos de notificação de eventos. No entanto, mecanismos de notificação podem ser implementado através do redirecionamento de portas do *chipset* e do controlador de interrupções (*Advanced Programmable Interrupt Controller*—APIC) para que eventos gerem interrupções do modo SMM (SMIs), disparando tratadores de eventos dentro da BIOS instrumentada. A notificação de eventos através de interrupções também ocorre para contadores de *performance*. Neste caso, contudo, interrupções ordinárias, e não SMIs, são geradas por padrão. Quando da ocorrência da interrupção, uma rotina de tratamento a nível de *kernel* (*Interrupt Service Routine*—ISR) é responsável por coletar os valores armazenados nos registradores ou nas páginas de memória do sistema. Os demais mecanismos de monitoramento, tais como DMA, são passivos e, portanto, não geram interrupções.

3.4.2. Callbacks

Além do disparo de rotinas de inspeção quando da ocorrência de eventos, pode-se também executar código de monitoração e verificação em meio as instruções do programa analisado, através de mecanismos de *callbacks*. O uso de *callbacks* por mecanismos de análise está principalmente associado a soluções baseadas no modo SMM. Soluções SMM inserem *callbacks* em instruções, por exemplo, para superar as barreiras de endereçamento de memória física. Além disso, *callbacks* são utilizadas pelo modo SMM para causar *VM-EXITs* de um *hypervisor* em modo *root*, possibilitando, assim, a inspeção deste.

3.4.3. Monitoração de memória

O monitoramento de memória é uma tarefa de inspeção essencial, dado que os valores em memória permitem reconstruir o contexto da execução. De um modo geral, a monitoração de memória é implementada através da geração proposital de *page faults*. Nesta técnica, uma ou mais páginas são marcadas como não presentes ou sem a requerida permissão (de execução, leitura ou escrita), causando, portanto, um *page fault* durante uma tentativa de acesso. O sistema de gerenciamento de memória é instrumentado de modo a registrar o evento (*logging*) e restaurar a permissão retirada, permitindo, assim, que a execução prossiga. A escolha de qual permissão será retirada determina o tipo de evento monitorado. Uma política frequentemente imposta é a chamada `Write Xor Execute` [Roemer et al. 2012], que garante que páginas de código (executáveis) não podem ser escritas e/ou que páginas escritas (dados) não sejam executáveis. Desta forma, as tentativas de se escrever em páginas executáveis ou de se executar páginas com permissão de escrita resultam na violação da política, com respectiva invocação do tratador de *page faults*.

A sucessiva escrita e execução de código é observada, por exemplo, em ataques de *buffer overflow*, no qual código externo é injetado e executado a partir da pilha. Sistemas modernos impõem a política de pilha não executável, como mostrado no Código 3.3.

```
1 cat /proc/self/maps
2 00400000-0040c000 r-xp 00000000 /bin/cat
3 7f0bef204000-7f0bef3c4000 r-xp  libc-2.23.so
4 7ffe3a213000-7ffe3a234000 rw-p  [stack]
```

Código 3.3. Pilha não executável. Plataformas modernas não permitem que a pilha seja executada para evitar ataques de injeção de código. Note que apenas atributos de leitura e escrita (rw-) são atribuídos a pilha (*stack*).

A escrita e execução de páginas de código também é observada em exemplares de *malware* do tipo auto-modificável (*Self Modifying Code*—SMC) [Botacin et al. 2018a, Cai et al. 2007]. Neste tipo de exemplar, o código original se auto-modifica de modo a se tornar malicioso. Esta estratégia é empregada para que o código original possa sobreviver a análises por soluções de segurança, como por antivírus. Um exemplo de código auto-modificável é apresentado na Figura 3.17.

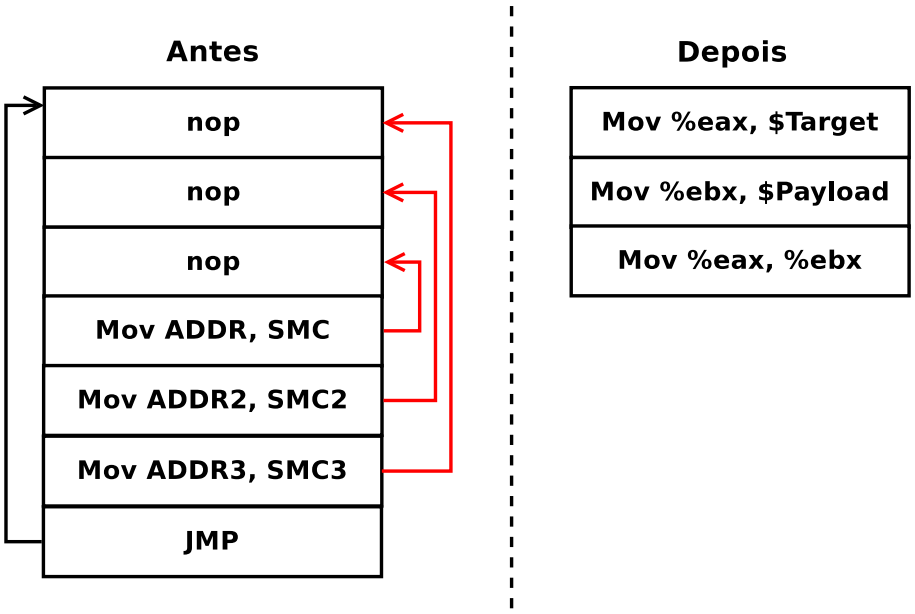

Figura 3.17. Código Auto-Modificável (SMC). Durante sua execução, o código altera suas próprias instruções (`Mov SMC`), tornando as instruções `NOP`, em instruções `MOV` de propósito malicioso. Após as modificações, o código salta (`JMP`) para poder executar as instruções modificadas.

Durante o tratamento do *page fault*, os dados presentes na página podem ser alterados pela solução de análise, o que é um requisito de soluções de *debugging*. Contudo, a forma usual de se implementar modificações nas páginas de memória é através de *shadow copies*. Esta técnica consiste em duplicar o conteúdo das páginas acessadas de modo a poder comparar a versão original e a versão modificada. Uma sequencia de *shadow copies* permite, por exemplo, realizar o *unpack* de códigos maliciosos. A cópia de valores de memória é uma tarefa constante durante processos de monitoração. Procedimentos forenses, por exemplo, realizam o *dump* da memória para análise *offline*. Verificadores de sistema, por sua vez, inspecionam a memória periodicamente para calcular sua integridade. Contudo, copiar a memória de forma completa é um procedimento custoso,

tanto em armazenamento quanto em tempo de processamento, o que requer abordagens alternativas. A técnica usual para este tipo de tarefa consiste em se realizar um primeiro *snapshot* completo do sistema e posteriormente acessar apenas as páginas modificadas. Os recursos arquiteturais como os controladores de memória permitem marcar as páginas modificadas e realizar a cópia apenas destas. Mais especificamente, pode-se postergar a cópia dos dados até que uma requisição para que estes sejam modificados seja feita, uma abordagem conhecida como *dump on write*, em referência a política *copy on write* dos sistemas Unix [Thober et al. 2008]. Este tipo de cópia de memória é ilustrado na Figura 3.18.

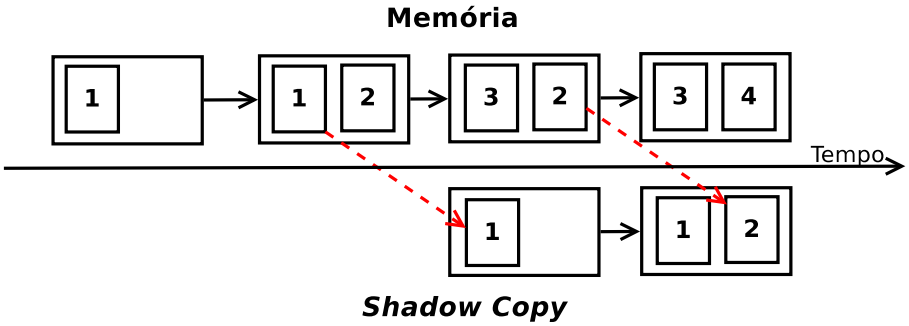

Figura 3.18. *Shadow Copies).* A duplicação de memória pode ser implementada de uma forma sob demanda, coletando os dados apenas quando estes são modificados, uma política denominada *dump-on-write*.

Enquanto as técnicas de manipulação de permissões e de *shadow copies* não são inéditas por si só, o uso destas foi impulsionado pelos novos recursos de MMU presentes nas arquiteturas modernas, notavelmente em HVMs.

3.4.4. Execução em pequenos passos

A execução em pequenos passos é uma tarefa de inspeção importante pois permite ao analista observar regiões de código específicas, com controle de granularidade. De um modo geral, o controle do avanço da execução em plataformas modernas pode ser implementado de duas maneiras: i) pelo controle das permissões das páginas de código; ou ii) por interrupções dos contadores de eventos. No primeiro caso, as páginas de código são marcadas como não-presentes e/ou não-executáveis, causando *page-faults* quando da tentativa de execução, requerindo que o tratador de eventos restaure a permissão da página para que a execução prossiga. Ao se repetir este procedimento para cada instrução, pode-se implementar o avanço de forma *step-by-step*, tal qual requerido por *debuggers*. No caso dos contadores de eventos, estes são pré-configurados em seus valores máximos, causando um *overflow* a cada instrução executada, dado que um novo evento de *hardware* é gerado. O tratador de interrupções é responsável por tratar o evento e novamente definir o contador para o valor máximo permitido, o que gerará uma nova interrupção quando da execução da instrução seguinte. Esta estratégia também permite o avanço *step-by-step*.

3.4.5. Tratamento de instruções individuais

Uma desvantagem das técnicas de interceptação baseadas em *faults* e *overflows* é que o tratador de eventos não tem acesso imediato a instrução que gerou o evento. O tratador

de *page faults*, por exemplo, tem acesso apenas ao endereço base da página. Para tratar estes casos, estes tratadores devem operar em conjunto com outros mecanismos, como os monitores de *branch*. Desta forma, quando o tratador de eventos é invocado, este deve olhar o topo da pilha de *branches* para identificar o último bloco de instruções executado.

3.4.6. Pontos de parada

Breakpoints são extensões naturais da execução em múltiplos passos, permitindo ao analista interromper a execução em pontos de interesse específicos. Processadores tradicionalmente apresentam suporte para dois tipos de *breakpoints*: *hardware breakpoints* e *software breakpoints*. *Hardware breakpoints* são registradores nos quais o analista pode inserir endereços de instruções para que o processador interrompa a execução quando o apontador de instruções (*Instruction Pointer*—IP) corresponder a um endereço de um dos registradores. Por requerir o armazenamento em registradores físicos, o número de *hardware breakpoints* é naturalmente limitado. Além disto, estes são compartilhados entre o sistema *host* e o sistema *guest*. *Software Breakpoints*, por sua vez, são ilimitados, mas introduzem efeitos colaterais de execução, pois requerem que os primeiros *bytes* da instrução monitorada sejam substituídos por uma *trap flag* (`int 3` ou `0xCC`), que faz com que o processador interrompa a execução quando esta é identificada.

A necessidade de se prover suporte a um grande número de pontos de parada e, ao mesmo tempo, não impor efeitos colaterais de execução, fez com que as soluções de análise modernas se baseassem em outras formas de se interromper a execução. De um modo geral, duas técnicas são frequentemente implementadas: i) execução por avanço-e-comparação; e ii) pontos de paradas invisíveis. No primeiro caso, as técnicas de avanço previamente apresentadas (*page fault trapping* e *performance counter overflows*) são empregadas de modo que o endereço do apontador de instruções seja comparado com uma lista de endereços de parada a cada etapa, consistindo, portanto, em uma implementação mista entre *hardware* e *software*. No segundo caso, *software breakpoints* tradicionais são utilizados, mas os efeitos colaterais são mascarados por tratadores de exceção. Por exemplo, a instrução `PUSHF` precisa ser interceptada para que a *trap flag* seja removida. Caso contrário, um programa monitorado poderia ler suas próprias *flags* e identificar que este está sob monitoração.

3.4.7. Filtragem de granularidade

Se, por um lado, a visão do sistema como um todo (*system-wide view*) permite ao analista trabalhar com modelos de ameaça abrangentes, como os que incluem o *kernel* ou o *hypervisor*, por outro, a quantidade de dados coletados pode dificultar o trabalho de inspeção. Por isso, filtros são empregados de modo a isolar eventos e regiões de interesse.

Uma escolha de granularidade frequente por parte dos analistas é quão profundo será o mergulho dentro das chamadas de funções. Uma visão de todo do sistema permite mergulhar recursivamente dentro das chamadas que implementam uma dada função de alto nível. Este tipo de operação é denominado *step-into* e é exemplificado pela Figura 3.19.

Podemos observar que, neste modo, as funções internas à função `printf` também são monitoradas, permitindo ao analista compreender a forma como a função é im-

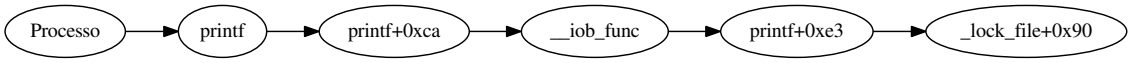

Figura 3.19. *Step-Into.* Quando a análise ocorre neste nível de granularidade, o analista mergulha em cada chamada de função, incluindo as funções auxiliares dentro da função principal sendo analisada. No exemplo, pode se observar os *locks* usados para garantir a sincronia da função `printf`.

plementada, o que é usual em processos de depuração e engenharia reversa. Identifica-se, por exemplo, o uso de *locks* para acesso exclusivo, dado que a função não é reentrante.

No entanto, quando o analista está interessado em informações mais gerais, como identificar o comportamento de um *malware* através das funções chamadas, este pode limitar o escopo da análise as chamadas de mais alto nível, desconsiderando as chamadas internas, o que é conhecido como modo *step-over*, como representado na Figura 3.20.

Figura 3.20. *Step-Over.* Quando a análise ocorre neste nível de granularidade, o analista está interessado nas funções de mais alto nível e, portanto, mais externas, simplificando o procedimento de análise. No exemplo, as chamadas internas das funções não são consideradas.

Neste exemplo, as chamadas internas das funções `printf` e `scanf` são omitidas, de modo que o analista pode identificar o comportamento geral da aplicação (ler um dado e imprimi-lo).

Dado o *gap* semântico, filtros de alto nível não são nativos do *hardware* e devem ser implementados pela solução de análise. De um modo geral, mecanismos de filtragem se baseiam em *breakpoints* condicionais controlados por *software*. Neste tipo de *breakpoint*, a parada da execução em um dado ponto acorre apenas quando uma condição de análise definida pelo analista é satisfeita, como, por exemplo, quando um dado registrador ou valor de memória assume o valor zero.

3.4.8. Mitigação de evasão por medidas de tempo

Apesar de não requerir injeções de código nem causar efeitos colaterais de execução, a análise de objetos em soluções de *hardware* ainda pode ser identificada por medidas de tempo. Notavelmente no caso de HVMs, o registrador *TimeStamp Counter* (TSC) é incrementado tanto para as instruções do *guest* quanto do *host*, o que permitiria a um *guest* malicioso identificar as instruções executadas pelo *hypervisor* instrumentado. Para lidar com este cenário, as soluções de análise modernas adulteram o valor do registrador TSC para omitir o tempo gasto pelo processamento no *hypervisor*. Por esta razão, medidas de tempo tomadas em máquinas virtuais, de um modo geral, não são confiáveis.

3.4.9. Mitigação de *fingerprinting*

Uma maneira de se detectar um ambiente de análise é localizar identificadores únicos conhecidos, como o nome de soluções de análise ou caminhos de sistema de soluções de segurança. De um modo geral, soluções de análise modernas empregam técnicas para evi-

tar este tipo de evasão. Em particular, soluções baseadas em HVM utilizam-se de técnicas de *rootkits* para esconder seus *drivers* de carregamento das APIs do sistema operacional. Soluções baseadas em SMM, por sua vez, geram SMIs baseadas em um *timer* periódico para executar rotinas de verificação, dentre as quais, se os registradores de *debug* foram modificados.

3.5. Aplicações

Nesta seção, apresentamos diversas aplicações de análise que podem se beneficiar dos recursos arquiteturais e das técnicas de implementação apresentadas.

3.5.1. Detecção de *bugs*

Defeitos de *software* (*bugs*) são a raíz de muitos problemas de segurança, o que torna a detecção destes tarefa fundamental. Embora diferentes soluções de monitoração possam ser aplicadas à esta tarefa, as soluções baseadas em monitores de *branch* são as mais adequadas, visto que a detecção de *bugs* exige apenas a coleta de dados relativos aos caminhos percorridos pela execução.

Programas podem ser entendidos como máquinas de estados, cujas transições são as instruções de *branch*. Um código compilado apresenta diversas instruções de *branch*, como apresentado no Código 3.4, de modo que os caminhos formados por estas representam diferentes estados. Um *bug* em um programa pode ser entendido como um estado inesperado, atingido através de uma sequência de *branches*. Portanto, identificar o *branch* origem do caminho até o estado inesperado permite identificar a causa do *bug*, como mostrado na Figura 3.21.

```
1 je      0x4b9cf6   0x4b9c40
2 jne     0x4b9c46   0x4b9bd1
3 jle     0x4b9bd3   0x4b9c88
4 jne     0x4b9c9c   0x4b9c88
5 je      0x4b9c9c   0x4b9c74
6 jne     0x4b9c7a   0x4b9cd8
7 je      0x4b9ce8   0x4b9ad0
8 je      0x4b9ae3   0x4b9b08
```

Código 3.4. *Disassembly*. Endereços (origem de destino) das instruções de controle de fluxo de uma aplicação que apresenta um *bug*.

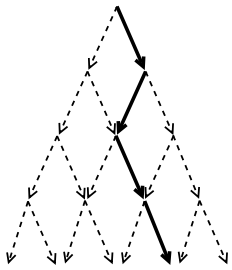

Figura 3.21. Árvore de decisão. O acompanhamento dos *branches* tomados permite identificar o caminho percorrido e, assim, identificar em qual ponto a execução diverge do esperado.

De um modo geral, a identificação de *bugs* pode ocorrer em três diferentes estágios do ciclo de vida de uma aplicação: i) durante o desenvolvimento; ii) durante o *deployment*; e iii) análises *post-mortem*.

Análises em tempo de desenvolvimento consistem em exercitar os diferentes caminhos de execução possíveis, em um procedimento denominado *fuzzing* [Felderer et al. 2016]. Para mapear os diferentes caminhos tomados, monitores de *branch* como o BTS podem ser empregados [Paleari 2015].

Análises de aplicações *deployed* consistem em monitorar o caminho de execução tomado pela aplicação quando exercitada com uma entrada que leve a uma falha. A monitoração dos caminhos visa identificar qual bloco de decisão leva a falha para sua posterior correção. Estratégias de *root-cause-analysis* podem, também, ser implementadas utilizando-se de monitores de *branch* como o LBR [Arulraj et al. 2014].

Análises *post-mortem* consistem em coletar dados de execução e submetê-los aos desenvolvedores na forma de *crash reports*. As informações de caminhos tomados podem ser enviadas para auxiliar na localização da falha de implementação. Esta abordagem pode ser implementada, por exemplo, através do uso do monitor `Processor Tracer` [Xu et al. 2017].

3.5.2. Análise de *Malware*

HVM e SMM são candidatos naturais para a implementação de soluções de análise de *malware*, pois eles permitem a execução do exemplares malicioso de forma transparente e livre de injeção de código. Além disso, a base de código confiável destes permite modelos de ameaça amplos, cobrindo, por exemplo, a análise de *rootkits* de *kernel*.

Diversas soluções baseadas em HVM foram propostas para a tarefa de análise de *malware* [Dinaburg et al. 2008b, Willems et al. 2012b, Nguyen et al. 2009]. De um modo geral, o recurso de *page fault trapping* é estendido para todo o sistema, de modo que qualquer parte deste tocada pelo exemplar de *malware* cause um `VM-EXIT`, a ser tratado pelo *hypervisor*. Este, por sua vez, é instrumentado de modo a implementar a solução de análise em si.

Uma desvantagem significativa do uso de HVM é o impacto no desempenho imposto pelas sucessivas saídas do *hypervisor*. Como alternativa, soluções de *replay* podem ser implementadas [Yan et al. 2012]. Neste tipo de abordagem, apenas a coleta dos dados de execução dos exemplares é realizada em um ambiente HVM, enquanto os procedimentos de análises são realizados em etapas posteriores, que podem ser implementados em *software*. Desta forma, obtém-se tanto as vantagens da execução em ambiente HVM—a mitigação das tentativas de evasão por parte do exemplar de *malware*—quanto a flexibilidade de uma solução em *software*. Este tipo de abordagem permite, por exemplo, repetir as instruções executadas pelo exemplar de *malware* em uma solução HVM em uma solução de *software*, de modo a identificar padrões maliciosos em sua execução.

O impacto no desempenho pode também ser mitigado pela adoção de tecnologias de coleta de dados mais leves, como os monitores de *branch* [Willems et al. 2012a, Botacin et al. 2018a]. Neste caso, contudo, restrições ao modelo de ameaças são impostas. Como os monitores requerem que o *kernel* trate as interrupções geradas por estes, este deve ser incluído na base de código confiável.

3.5.3. *Debugging*

Mais do que traçar o comportamento de programas, os recursos arquiteturais das plataformas modernas permitem também interromper a execução do objeto monitorado em pontos de interesse e alterar o contexto deste (incluindo valores em memória, em registradores e até mesmo o apontador de instruções), o que os torna adequados para a implementação de ferramentas de *debugging*.

Soluções baseadas em HVM [Fattori et al. 2010] são capazes de prover *stepping* de instruções, definição de pontos de paradas e acesso a registradores e memória através de `VM-EXITs`. Além disso, a capacidade de atuar no sistema como um todo possibilita inclusive o *debugging* do *kernel* do sistema em execução. Por esta mesma característica, soluções de *debugging* baseadas em SMM também foram propostas [Zhang et al. 2015]. Os recursos de inspeção providos por esta são similares aos das soluções HVM.

As interrupções geradas por contadores de *performance* também podem ser utilizadas para gerar pontos de parada para *debugging*. Contudo, seu tratamento a nível de *kernel* permite apenas que aplicações do modo usuários sejam inspecionadas. Além disso, o tipo de monitor utilizado influencia na granularidade da inspeção. O uso de monitores de *branch*, por exemplo, permite que a execução ocorra *branch-by-branch* ao invés de *step-by-step* [Botacin et al. 2018a].

3.5.4. Forense

A capacidade de executar código privilegiado também torna os recursos das arquiteturas modernas adequados para a extração de dados do sistema para fins de forense.

Soluções baseadas em HVM [Martignoni et al. 2010] são capazes de ser instanciadas em ambientes já em execução devido ao recurso *live loading*, sendo adequadas para o uso por peritos, como mostrado na Figura 3.22. Como o ambiente pode ter sido comprometido, a solução emprega um mecanismo de *lie detection* para verificar a confiança do sistema. Neste, processos em execução são obtidos através das APIs do SO e também via introspecção do *hypervisor*. O sistema é considerado comprometido se os dados em ambas as fontes divergirem.

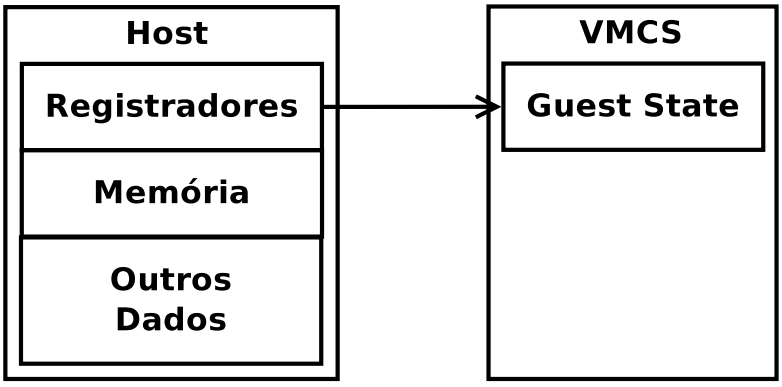

Figura 3.22. Carregamento em tempo real. Soluções HVM permitem copiar o estado corrente da máquina física para o contexto de uma máquina virtual, movendo o ambiente em tempo de execução. Este recurso permite, por exemplo, que analises forenses sejam realizadas utilizando-se da própria máquina a ser analisada.

Soluções forenses baseadas em SMM também foram propostas [Reina et al. 2012]. Como um mecanismo de *live loading* nativo não existe neste modo, a BIOS instrumentada deve ser instalada *a priori*. Como, de um modo geral, não se sabe a tarefa forense a ser requerida de antemão, usualmente se utiliza o modo SMM para realizar o *dump* total da memoria do sistema para que as análises sejam realizadas posteriormente.

Uma forma leve, em termos de desempenho e de implementação, de se realizar o *dump* completo da memória do sistema é utilizar-se do acesso DMA, como proposto por algumas soluções [Wang et al. 2011]. Neste caso, o perito poderia inserir uma placa PCI no sistema e utilizar do acesso DMA para varrer a memória como um todo. Para garantir a consistência, isto é, que os dados não sejam alterados durante o processo de coleta, o sistema deve ser pausado de alguma forma.

3.5.5. Imposição de políticas de segurança

Além de permitir aos analistas inspecionarem os binários em execução, os recursos arquiteturais podem ser empregados para impor políticas de execução sobre estes em tempo real. Isto garante a organizações, por exemplo, que suas políticas de segurança sejam impostas, a despeito de recursos da aplicação ou do sistema operacional, uma vez que a imposição é implementada em camadas inferiores.

O modo HVM, por exemplo, pode ser utilizado para impor politicas de armazenamento, como proposto em Bitvisor [Shinagawa et al. 2009]. Nesta solução, todas as escritas em disco são interceptadas e criptografadas, garantindo a segurança dos dados armazenados, dado que a chave de decriptação fica armazenada no *hypervisor*, a qual o *guest* não tem acesso.

3.5.6. Detecção de ataques

Os recursos arquiteturais das plataformas modernas também podem ser utilizados para detectar ataques em tempo real, se beneficiando dos recursos de análise suportados por estes.

Soluções baseadas em HVM podem causar `VM-EXITs` quando eventos suspeitos ocorrem. A solução Secvisor [Seshadri et al. 2007], por exemplo, permite que apenas código aprovado seja executado, invocando as rotinas de inspeção do *hypervisor* quando de violações desta política. Este tipo de política impede, por exemplo, ataques de redireção do *kernel* para o modo usuário. Neste tipo de ataque, o atacante se vale dos privilégios de execução do *kernel*, através, por exemplo, de um *bug* em um *driver*, para escrever em processos de modo usuário, estejam estes sob controle do atacante ou de terceiros. Se esta escrita contemplar *flags* de proteção, o processo afetado pode ter seus mecanismos de segurança desabilitados através da atribuição de privilégios administrativos.

As capacidades do modo SMM também podem ser utilizadas para a detecção de ataques. A solução SPECTRE [Zhang et al. 2013], por exemplo, implementa verificações em memória em tempo real. Pode-se, por exemplo, aplicar expressões regulares sob os dados coletados para se detectar a injeção de *shellcodes* através de uma sequência de NOPs (*No-Operation*), uma característica de ataques do tipo *buffer overflow*, dada a necessidade de alinhamento do código na pilha.

Além de políticas de detecção de ataques de um modo geral, alguns recursos arquiteturais são particularmente interessantes para a detecção de ataques específicos, como as políticas de integridade do fluxo de controle e a detecção por efeitos colaterais. Atualmente, ataques de injeção de código utilizam da técnica denominada programação orientada a retorno (*Return Oriented Programming*—ROP), de modo a contornar políticas de restrição da execução de código externo, como `Write Xor Execute`. Neste tipo

de ataque, ao invés de injetar um *shellcode* externamente, como em ataques de *buffer overflow* tradicionais, o *shellcode* é construído em memória através do encadeamento de sequências de instruções terminadas por uma instrução de retorno (`RET`) (*gadgets*), como mostrado na Figura 3.23.

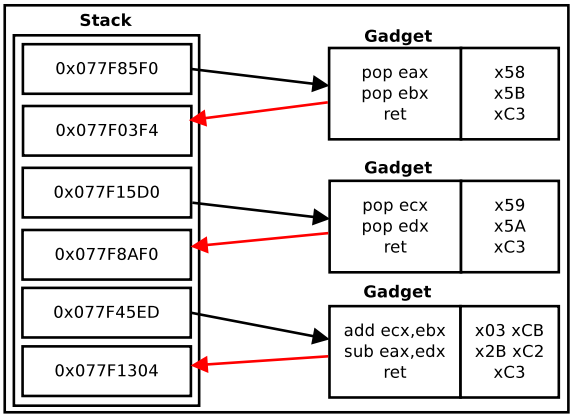

Figura 3.23. Programação Orientada a Retorno (ROP). O *payload* neste tipo de ataque é construído através do encadeamento de sequências de instruções terminadas por uma instrução de retorno (*gadgets*).

Este tipo de construção de código viola uma expectativa comum de que toda instrução de retorno seja precedida por uma instrução do tipo `CALL`, uma vez que chamadas de função seguem este padrão. Desta forma, ataques `ROP` podem ser detectados por políticas de integridade do fluxo de controle, como garantir que todo bloco de instruções terminados em `RET` seja precedido por uma instrução `CALL`, como exemplificado na Figura 3.24.

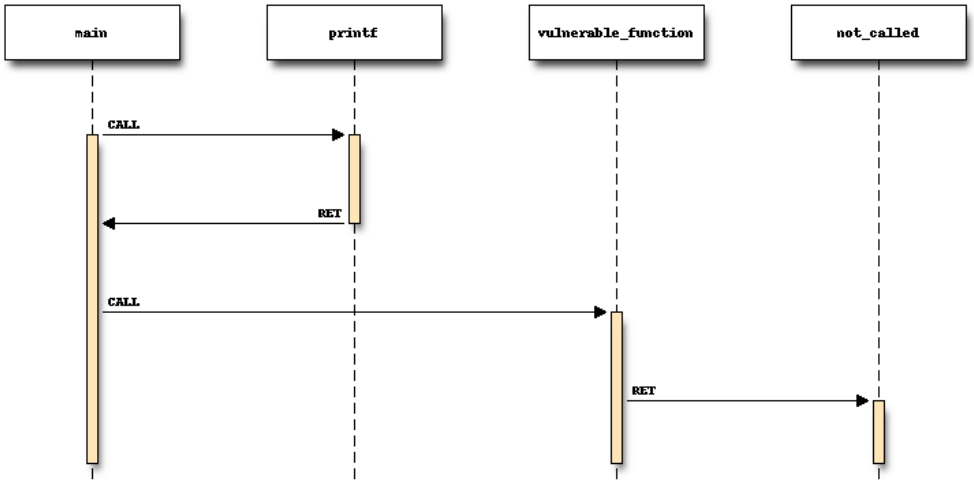

Figura 3.24. Integridade de Fluxo de Controle (CFI). Tipicamente, espera-se que instruções de retorno sejam precedidas por instruções do tipo `CALL` (como no fluxo entre *main* e *printf*). Em um ataque do tipo ROP, esta condição é violada, o que permite sua identificação quando da execução do *payload* (como no fluxo entre *vulnerable_function* e *not_called*).

Políticas de integridade do fluxo de controle (*Control Flow Integrity*—CFI) podem ser implementadas de forma eficiente através do uso de monitores de *branches*, dado que estes podem monitorar especificamente este tipo de instrução, tal qual proposto em diversas soluções [Xia et al. 2012, Pappas et al. 2013, Cheng et al. 2014]. Por monitorar apenas as instruções de desvio de fluxo, este tipo de abordagem reduz significativamente o impacto da solução de monitoração sobre o sistema em comparação com soluções baseadas em *stepping*, que necessitam verificar todas as instruções para posteriormente desconsiderar as que não estejam envolvidas na mudança do fluxo de execução.

Além de políticas explicitas, como CFI, políticas implícitas também podem ser empregadas para a detecção de ataques. A detecção de efeitos colaterais consiste em se definir um padrão normal de execução do sistema, tais como o número de instruções de *branch* executadas em um intervalo de tempo, e comparar os resultados obtidos em tempo real com este padrão normal. Desta forma, ataques `ROP` seriam detectados pela execução anômala de seguidas instruções de retorno. Contadores de *performance* são adequados para esta tarefa, tal qual proposto em [Kompalli and Sarat 2014, Bahador et al. 2014], dado que estes podem contar eventos no sistema como um todo sem a necessidade de se instrumentar cada aplicação.

3.5.7. Verificação de integridade

Além de tentar detectar ataques de forma deliberada, os recursos arquiteturais podem ser utilizados para garantir a integridade do sistema e assim prover garantias de segurança derivadas desta.

Tipicamente, soluções de verificação de integridade são implementadas com o suporte de *hardware* externo [Petroni et al. 2004, Moon et al. 2012, Lee et al. 2013]. Este tipo de solução é capaz de efetuar a leitura da memória a ser verificada via barramento compartilhado e computar um *hash* dos valores lidos, identificando se estes se alteram ao longo do tempo. A hipótese que suporta esta abordagem é que alguns componentes do sistema, como o código de *boot* ou do *kernel*, devem ser imutáveis.

Soluções HVM também podem ser utilizados para verificar a integridade de componentes do sistema, como as configurações de I/O [Zhang 2013], de modo a garantir que as portas não sejam remapeadas. O remapeamento de mecanismo de DMA poderia levar, por exemplo, a um ataque do tipo *man-in-the-middle* de memória, dando ao atacante controle sob os dados.

A depender do modelo de ameças considerado, verificações a nível de *hypervisor* podem não ser adequadas, visto que ataques aos *hypervisors* são atualmente conhecidos [Rutkowska and Wojtczuk 2008]. Neste tipo de ataque, o código em execução dentro da máquina virtual mapeia a memória física do *hypervisor* de modo a fazer com que este atribua maiores permissões ao código monitorado, como exemplificado pela Figura 3.25.

Desta forma, garantir a integridade do *hypervisor* em si é uma tarefa essencial em muitos cenários para prevenir e detectar este tipo de modificação. Para tanto, soluções em mais baixo nível são necessárias, tais como as baseadas no modo SMM [Azab et al. 2010, Wang et al. 2010]. Tal qual as abordagens em *hardware* externo, estas soluções também se baseiam em *hashes* para garantir que o código do *hypervisor* não seja alterado.

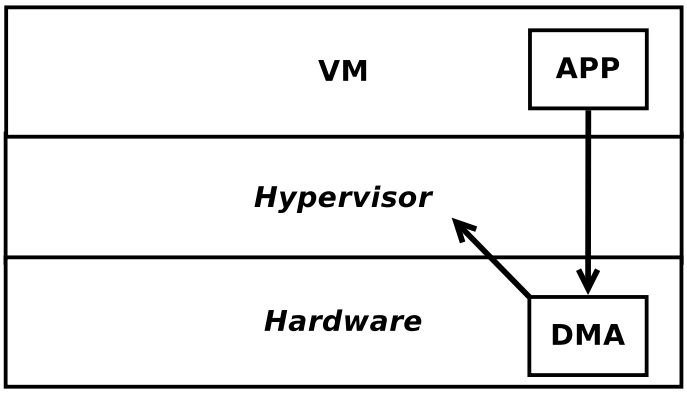

Figura 3.25. Ataques ao *Hypervisor*. A aplicação dentro da máquina virtual pode mapear a memória física do *hypevisor* e, através de acesso DMA, modificar seu conteúdo. A aplicação pode, por exemplo, alterar as permissões que o *hypervisor* atribui a sua execução, elevando, assim, seus privilégios.

3.6. Perspectivas Futuras: Ameaças e Oportunidades

Os recursos arquiteturais presentes nas arquiteturas modernas trouxeram novas possibilidades de implementação de mecanismos de análise e também novos desafios. Esta seção apresenta os problemas atualmente em aberto e perspectivas de futuros desenvolvimentos. Acreditamos que os pontos aqui apresentados podem ser abordados em futuros projetos de pesquisa.

Máquinas Virtuais de *Hardware* apresentam um enorme potencial para o desenvolvimento de soluções de segurança e por isso novas aplicações tem se baseado fortemente neste tipo de recurso. O sistema operacional Qubes [Rutkowska 2010], por exemplo, move todo o ambiente Linux para uma máquina virtual de modo a isolar os processos em execução através do *hypervisor*. Tentativas de violar o controle de acesso resultam em exceções do *hypervisor*. Estratégia similar é empregada no sistema Windows 10 quando da utilização do modo *Secure Kernel* [Ionescu 2015]. Com a consolidação desta tendência, técnicas de detecção de máquina virtual não serão suficientes para inferir um ambiente como sendo de análise. Desta forma, códigos evasivos terão de diretamente detectar que estão sob análise, utilizando-se, por exemplo, de informações do contexto da execução, o que é significativamente mais difícil. Entre os desafios para a consolidação das máquinas virtuais de *hardware* estão o suporte a *guests* aninhados (máquinas virtuais dentro de máquinas virtuais) e o desenvolvimento de processos de introspecção para este cenário.

A consolidação dos ambientes virtualizados pode, por outro lado, permitir novos recursos de segurança para as aplicações. Aplicações conscientes de sua execução em uma máquina virtual podem delegar ao *hypervisor* diversas tarefas. A delegação ocorre através de chamadas do *hypervisor* (*hypercalls*) a partir do *kernel*. *Hypercalls* podem ser entendidas como uma extensão das *syscalls* que ocorrem do modo usuário para o *kernel* (Figura 3.26). Soluções atuais, como o Xen, já fornecem algum suporte, ainda que limitado, a *hypercalls*, como mostrado no Código 3.5. Este tipo de chamada pode ser estendida para que um *kernel* notifique o *hypervisor* sobre, por exemplo, uma região de memória que deve ser protegida por este.

O **modo SMM** pode ser empregado em diversas aplicações de segurança, tendo

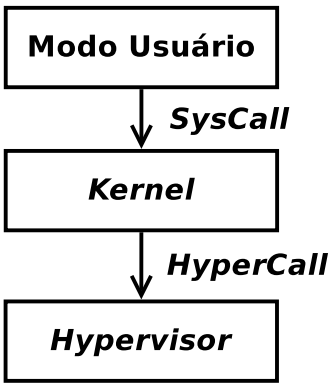

Figura 3.26. *Hypercall.* Chamadas do *hypervisor* a partir do *kernel* são correspondentes a chamadas deste pelo modo usuário.

```
1 enum hypercall_num {
2 #define __HYPERVISOR_set_callbacks
3 #define __HYPERVISOR_set_debugreg
4 #define __HYPERVISOR_get_debugreg
5 #define __HYPERVISOR_xen_version
6 #define __HYPERVISOR_vm_assist
7 #define __HYPERVISOR_callback_op
8 #define __HYPERVISOR_sysctl
9 #define __HYPERVISOR_domctl
```

Código 3.5. Código do Xen. *Hypercalls* são suportadas pelo *hypervisor*, de forma que um *kernel* ciente da sua operação em uma máquina virtual pode invocá-las para assistir sua operação.

notável sucesso na verificação de integridade de *hypervisors*, dado seu posicionamento privilegiado. Como desafio, tal qual para HVMs, procedimentos de introspecção devem ser desenvolvidos para que o modo SMM possa interpretar os conteúdos em execução dentro do *hypervisor*. Além disso, a instrumentação de BIOS não é uma tarefa simples e pode ser até mesmo impedida em algumas plataformas recentes.

A aplicação dos modos **ME/AMT** evoluiu de mecanismos de monitoração e controle do estado do processador, tais como consumo de energia e temperatura, para a monitoração da execução em si, tendo como maior vantagem possibilitar o controle total sobre o sistema. Atualmente, apenas provas de conceitos utilizando-se deste modo foram apresentadas, mas acreditamos que esta tecnologia possui potencial para ser aplicada em tarefas como *tracing* e *debugging*.

Contadores de *Performance* são uma alternativa leve para a monitoração de sistemas em tempo real. Enquanto soluções que identificam desvios de execução de um perfil do sistema definido a partir dos dados dos sensores são bem estabelecidas, avanços recentes das técnicas de aprendizados de máquina podem impulsionar novos desenvolvimentos. Abordagens leves como estas devem ser popularizadas a medida que as demandas por soluções de alta performance aumentem.

O **Acesso direto à memória** é uma abordagem poderosa para a monitoração do sistema de uma maneira ampla. Como desvantagem, o contexto limitado pela ausência de registradores dificulta o desenvolvimento de mecanismos de introspecção. Além disso, um problema em aberto relativo a DMA é a ausência de autenticação de dispositivos PCI [Intel 2018], que implementam os mecanismos DMA, o que permite a introdução de dispositivos maliciosos em um sistema. Um dispositivo PCI malicioso pode, por exemplo, realizar o *dump* de memória em busca de chaves criptográficas e outros segredos.

Dispositivos de *hardware* externos são exemplo de uma classe de aplicações pouco explorada por mecanismos de segurança. Este tipo de mecanismo apresenta significativas vantagens, como não impor nenhum *overhead* ao sistema e ser protegido de subversão pelo seu posicionamento externo. Este posicionamento, contudo, limita as

soluções ao papel de agentes passivos, sendo capazes apenas de reportar ataques mas não de bloqueá-los. A extensão destas soluções para mecanismos ativos é um problema em aberto.

Enclaves Isolados, tais como o modo SGX, surgiram para conter o vazamento de informações mesmo na presença de aplicações que executem em *rings* mais privilegiados, como HVM e SMM. Contudo, ao mesmo tempo em que este modo possibilita proteger aplicações legítimas, tais como gerenciadores de chaves criptográficas, de subversões e vazamentos, este modo também impede a inspeção de aplicações maliciosas.

Enquanto um binário SGX pode ser analisado estaticamente, fora do enclave, a sua execução dentro do enclave não pode ser monitorada devido ao isolamento das páginas de memória. Desta forma, atacantes podem criar exemplares que evadam a monitoração estática por parte de soluções de segurança, como antivírus, transferindo a exibição de seu comportamento malicioso para a etapa de execução dentro do enclave. Isto pode ser feito através da compressão do binário e/ou da geração de código em tempo de execução. Como o código gerado em tempo real fica armazenado nas páginas de memória, este é isolado pelo enclave de qualquer inspeção externa. Como agravante, este tipo de mecanismo pode ser usado para implementar exemplares do tipo *downloader* [Rossow et al. 2013], que obtém diferentes *payloads* através da Internet e os extrai apenas dentro do enclave. A adoção de conexão criptografada, como `HTTPS`, pode impedir completamente a inspeção do tráfego entre o enclave e o servidor remoto, tal qual proposto em exemplares conceituais [van Prooijen 2016]. Esse tipo de operação maliciosa é ilustrado na Figura 3.27.

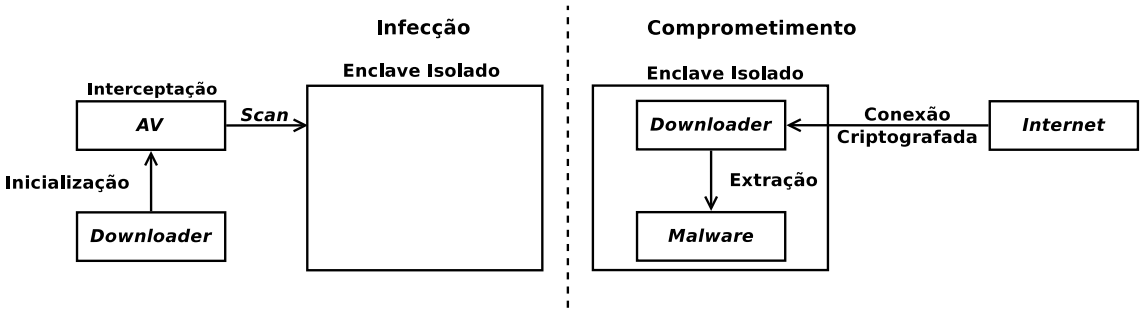

Figura 3.27. *Malware* em um enclave isolado. Atacantes podem evadir a detecção por antivírus movendo seus códigos para dentro de um enclave isolado. Ao se modularizar a amostra, o antivírus é capaz apenas de analisar o componente que carrega os demais, que não é malicioso por definição. Este componente, por sua vez, obtém os componentes maliciosos apenas dentro do enclave isolado, estando livre, portanto, de inspeções.

Atualmente, a única forma de se obter informações, ainda que limitadas, a partir de enclaves SGX é através de métodos de canal lateral [Schwarz et al. 2017]. Neste tipo de ataque, meta-informações de execução são utilizadas para inferir o conteúdo do enclave. Por exemplo, pode-se utilizar do monitor *Processor Tracer* para se comparar o perfil (frequência de *branches*, por exemplo) de uma aplicação conhecida quando em execução fora do enclave com o perfil da aplicação desconhecida em execução no enclave de modo a identificá-la [Lee et al. 2017]. Este tipo de técnica é efetiva visto que, apesar das páginas de memória serem isoladas, o modo SGX compartilha recursos de CPU, como

ALUs, unidades de *branch* e memória *cache* com as demais aplicações em execução no processador.

Ataques, de um modo geral, podem ser desenvolvidos utilizando-se não apenas de enclaves isolados, mas de todas as tecnologias anteriormente apresentadas, pois, da mesma maneira que as capacidades de monitoração destas podem ser empregadas para fins de análise, estas podem ser empregadas para propósitos maliciosos.

O carregamento de máquinas virtuais de *hardware* em sistemas já em execução (*live loading*) pode ser utilizado para mover sistemas inteiros para um estado infectado. possibilitando *rootkits* com ampla abrangência [Rutkowska 2006, Myers and Youndt 2007] e até mesmo com operação multi-plataforma. Além de HVMs, *rootkits* podem ser também implementados em SMM [Duflot et al. 2007, BSDaemon et al. 2008, Wecherowski 2009], se beneficiando da visão de todo o sistema que esse modo possibilita. O modo SMM pode ser utilizado, ainda, para a implementação de *keyloggers* [Embleton et al. 2008, Schiffman and Kaplan 2014], através do redirecionamento das interrupções de teclados para SMIs. *Keyloggers* podem ser também implementados a partir de dispositivos PCI, como GPUs [Ladakis et al. 2013], que se utilizem do acesso via DMA para monitorar escritas no *buffer* de teclado.

Enquanto tais ameaças ainda são, em sua maioria, provas de conceito, ataques utilizando-se destas possibilidades poderão ser visto na prática em um futuro próximo, tais como o primeiro *malware* de modo AMT foi recentemente identificado [Khandelwal 2017].

3.7. Considerações Finais

A análise de binários é uma tarefa fundamental dos processos de segurança, sendo aplicada em mecanismos de verificação e validação de códigos e sistemas. Arquiteturas modernas são complexas e, ao mesmo tempo em que possibilitam construções mais eficientes, trazem novos desafios para a monitoração dos sistemas construídos sobre elas. Notavelmente, a interferência em aplicações em execução é significantemente dificultada por mecanismos arquiteturais de proteção. Se, por um lado, isto amplia a proteção de aplicações legítimas, por outro, dificulta a análise de binários maliciosos.

Neste capítulo, apresentamos um panorama dos elementos estruturais presentes nas arquiteturas modernas, de modo a destacar os benefícios advindos destes e também os novos desafios impostos pelos mesmos. Destacamos, por exemplo, que o uso de enclaves isolados é capaz de aumentar a resistência de aplicações protegidas contra ataques diretos ao mesmo tempo em que permite a criação de exemplares de *malware* resistentes a inspeção por soluções antivírus tradicionais.

Na Seção 3.2, apresentamos os principais recursos de inspeção presentes em soluções de análise modernas, tal qual o carregamento de soluções forense em sistemas infectados, técnica que se tornou possível somente com o desenvolvimento das extensões de máquinas virtuais de *hardware* presentes nas arquiteturas modernas.

Na Seção 3.3, apresentamos diferentes tecnologias e como estas podem ser utilizadas para implementar os distintos recursos de inspeção anteriormente apresentadas. Discutimos soluções de diferentes granularidades e mostramos que, enquanto soluções

como máquinas virtuais de *hardware* possuem suporte nativo a muitas das tarefas de inspeção, outras soluções, como as baseadas no modo SMM, necessitam de maiores desenvolvimentos.

Na Seção 3.4, apresentamos as técnicas de monitoração empregadas de modo a prover os recursos de inspeção anteriormente apresentados. Discutimos, por exemplo, como seguidos *overflows* de contadores de *performance* podem ser utilizados para implementar execução *step-by-step*. Este tipo de técnica é embasada pela compreensão dos efeitos causados pela introdução deste tipo de recurso (contadores) na plataforma.

Na Seção 3.5, apresentamos diferentes aplicações para as tecnologias e técnicas de inspeção anteriormente discutidas. Nossa abordagem cobriu soluções de tempo real, como detecção de ataques e políticas de integridade de fluxo, e de inspeção, como as voltadas para procedimentos forenses e *debugging*.

Na Seção 3.6, discutimos o atual estágio de desenvolvimento do campo de soluções de análise com suporte de *hardware* e apontamos problemas em aberto. Dentre esses, destacam-se os desafios impostos pela adoção de enclaves isolados, tais como SGX, que, por um lado, protege aplicações legítimas de subversão, mas, ao mesmo tempo, torna exemplares de código maliciosos mais difíceis de serem analizados em tempo de execução.

Acreditamos que este trabalho possa contribuir com estudantes e profissionais da área em seus desenvolvimentos, seja pela apresentação dos recursos presentes nas arquiteturas modernas, pela categorização destes de acordo com a respectiva tarefa de análise ou mesmo pela identificação dos problemas em aberto, que podem ser abordados pela audiência em seus futuros projetos de pesquisa.

Por fim, nós encorajamos a leitura do *survey* [Botacin et al. 2018b], que prove informações complementares sobre mecanismos e técnicas de análise com suporte de *hardware*.

Agradecimentos. Os autores agradecem a 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

[AMD 2013] AMD (2013). *AMD64 Architecture Programmer's Manual Volume 2*. AMD.

[AMD 2016] AMD (2016). Amd secure processor (built-in technology). `https://tinyurl.com/yaq2rhmv`.

[ARM 2009] ARM (2009). *ARM Sec. Technology - Building a Secure System using TrustZone Technology*. ARM.

[Arulraj et al. 2014] Arulraj, J., Jin, G., and Lu, S. (2014). Leveraging the short-term memory of hardware to diagnose production-run software failures. *SIGARCH Comput. Archit. News*, 42(1).

[Aumasson and Merino 2016] Aumasson, J. and Merino, L. (2016). Sgx secure enclaves in practice: Sec. and crypto review.

[Azab et al. 2010] Azab, A. M., Ning, P., Wang, Z., Jiang, X., Zhang, X., and Skalsky, N. C. (2010). Hypersentry: Enabling stealthy in-context measurement of hypervisor integrity. In *Proc. 17th ACM Conf. on Comp. and Comm. Sec.*, CCS '10. ACM.

[Bahador et al. 2014] Bahador, M., Abadi, M., and Tajoddin, A. (2014). Hpcmalhunter: Behavioral malware detection using hardware performance counters and singular value decomposition. In *2014 4th Intl. Conf. on Comp. and Knowledge Engineering (ICCKE)*.

[Barbosa and Branco 2014] Barbosa, G. N. and Branco, R. R. (2014). Prevalent characteristics in modern malware. `http://www.kernelhacking.com/rodrigo/docs/blackhat2014-presentation.pdf`.

[Bayer et al. 2006] Bayer, U., Kruegel, C., and Kirda, E. (2006). Ttanalyze: A tool for analyzing malware. In *15th European Inst. for Comp. Antivirus Research (EICAR 2006) Annual Conf.* EICAR.

[Bellard 2005] Bellard, F. (2005). Qemu, a fast and portable dynamic translator. In *Proc. USENIX Annual Technical Conf.*, ATC '05. USENIX Association.

[Botacin et al. 2017] Botacin, Falcão, Geus, and Grégio (2017). Analysis, anti-analysis, anti-anti-analysis: An overview of the evasive malware scenario. `https://sbseg2017.redes.unb.br/wp-content/uploads/2017/04/20171109_ANAIS_SBSEG_2017_FINAL_E-BOOK.pdf`.

[Botacin et al. 2015] Botacin, Geus, and Grégio (2015). Uma visão geral do malware ativo no espaço nacional da internet entre 2012 e 2015. `http://siaiap34.univali.br/sbseg2015/anais/WFC/artigoWFC02.pdf`.

[Botacin et al. 2018a] Botacin, M., Geus, P. L. D., and Grégio, A. (2018a). Enhancing branch monitoring for security purposes: From control flow integrity to malware analysis and debugging. *ACM Trans. Priv. Secur.*, 21(1):4:1–4:30.

[Botacin et al. 2018b] Botacin, M., Geus, P. L. D., and grégio, A. (2018b). Who watches the watchmen: A security-focused review on current state-of-the-art techniques, tools, and methods for systems and binary analysis on modern platforms. *ACM Comput. Surv.*, 51(4):69:1–69:34.

[Botacin et al. 2018c] Botacin, M. F., de Geus, P. L., and Grégio, A. R. A. (2018c). The other guys: automated analysis of marginalized malware. *Journal of Computer Virology and Hacking Techniques*, 14(1):87–98.

[Branco et al. 2012] Branco, R. R., Barbosa, G. N., and Neto, P. D. (2012). Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti- vm technologies. `http://www.kernelhacking.com/rodrigo/docs/blackhat2012-paper.pdf`.

[BSDaemon et al. 2008] BSDaemon, coideloco, and D0nad0n (2008). System management mode hack - using smm for "other purposes". https://tinyurl.com/jxeao4u.

[Cai et al. 2007] Cai, H., Shao, Z., and Vaynberg, A. (2007). Certified self-modifying code. *SIGPLAN Not.*, 42(6):66–77.

[Chen et al. 2008] 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 Intl. Conf. on Depend. Syst. and Net. With FTCS and DCC (DSN)*. IEEE.

[Cheng et al. 2014] Cheng, Y., Zhou, Z., Miao, Y., Ding, X., and Deng, H. R. (2014). Ropecker: A generic and practical approach for defending against rop attacks. In *Symp. on Net. and Dist. System Sec. (NDSS)*. Internet Society.

[CoreBoot 2015] CoreBoot (2015). Coreboot. http://www.coreboot.org/.

[Dinaburg et al. 2008a] Dinaburg, A., Royal, P., Sharif, M., and Lee, W. (2008a). Ether: Malware analysis via hardware virtualization extensions. In *Proc. of the 15th ACM Conf. on Comp. and Comm. Sec.*, CCS '08. ACM.

[Dinaburg et al. 2008b] Dinaburg, A., Royal, P., Sharif, M., and Lee, W. (2008b). Ether: Malware analysis via hardware virtualization extensions. In *Proc. 15th ACM Conf. on Comp. and Comm. Sec.*, CCS '08. ACM.

[do Brasil 2018] do Brasil, G. (2018). Governo digital. https://www.governodigital.gov.br/.

[Duflot et al. 2007] Duflot, L., Etiemble, D., and Grumelard, O. (2007). Using cpu system management mode to circumvent operating system sec. functions. https://tinyurl.com/y7mlduy9.

[DynamoRIO 2001] DynamoRIO (2001). Dynamic instrumentation tool platform. https://tinyurl.com/ybenfvw9.

[Embleton et al. 2008] Embleton, S., Sparks, S., and Zou, C. (2008). Smm rootkits: A new breed of os independent malware. In *Proc. 4th Intl. Conf. on Sec. and Priv. in Communication Netowrks*, SecureComm '08. ACM.

[Fattori et al. 2010] Fattori, A., Paleari, R., Martignoni, L., and Monga, M. (2010). Dynamic and transparent analysis of commodity production syst. In *Proc. IEEE/ACM Int. Conf. on Automated Software Engineering*, ASE '10. ACM.

[Felderer et al. 2016] Felderer, M., Büchler, M., Johns, M., D. Brucker, A., Breu, R., and Pretschner, A. (2016). Sec. testing: A survey.

[G1 2017] G1 (2017). Mobile banking se torna meio mais usado para transacoes bancarias, diz febraban. https://g1.globo.com/economia/seu-dinheiro/noticia/mobile-banking-se-torna-meio-mais-usado-para-transacoes-bancarias-diz-febraban.ghtml.

[Grégio et al. 2013] Grégio, A. R. A., Fernandes, D. S. o., Afonso, V. M., de Geus, P. L., Martins, V. F., and Jino, M. (2013). An empirical analysis of malicious internet banking software behavior. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1830–1835, New York, NY, USA. ACM.

[Intel 2013] Intel (2013). *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel.

[Intel 2015] Intel (2015). Pin - a dynamic binary instrumentation tool. https://tinyurl.com/m685m25.

[Intel 2018] Intel (2018). Pcie device security enhancements specification. https://www.intel.com/content/www/us/en/io/pci-express/pcie-device-security-enhancements-spec.html.

[Ionescu 2015] Ionescu, A. (2015). Battle of the skm and ium: How windows 10 rewrites os architecture. https://tinyurl.com/na375ur.

[ISECLAB 2010] ISECLAB (2010). Anubis - malware analysis for unknown binaries. https://anubis.iseclab.org/.

[Kang et al. 2009] Kang, M. G., Yin, H., Hanna, S., McCamant, S., and Song, D. (2009). Emulating emulation-resistant malware. In *Proc. 1st ACM Work. on Virtual Machine Sec.*, VMSec '09. ACM.

[Kaspersky 2015] Kaspersky (2015). Beaches, carnivals and cybercrime: Kaspersky lab shares insights on brazilian cyber underground. https://www.kaspersky.com/about/press-releases/2015_beaches-carnivals-and-cybercrime-kaspersky-lab-shares-insights-on-brazilian-cyber-underground.

[Khandelwal 2017] Khandelwal, S. (2017). First-ever data stealing malware found using intel amt tool to bypass firewall. https://tinyurl.com/y7e7kg8v.

[Kompalli and Sarat 2014] Kompalli and Sarat (2014). Using existing hardware services for malware detection. In *Proc. 2014 IEEE Sec. and Priv. Work.s*, SPW'14. IEEE Comp. Society.

[Ladakis et al. 2013] Ladakis, E., Koromilas, L., Vasiliadis, G., Polychronakis, M., and Ioannidis, S. (2013). You can type, but you can't hide: A stealthy gpu-based keylogger. https://tinyurl.com/cbzp42n.

[Lee et al. 2013] Lee, H., Moon, H., Jang, D., Kim, K., Lee, J., Paek, Y., and Kang, B. B. (2013). Ki-mon: A hardware-assisted event-triggered monitoring platform for mutable kernel object. In *22nd USENIX Sec. Symposium*. USENIX.

[Lee et al. 2017] Lee, S., Shih, M.-W., Gera, P., Kim, T., Kim, H., and Peinado, M. (2017). Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Sec. Symposium (USENIX Sec. 17)*. USENIX Association.

[Li et al. 2004] Li, K., Ding, S., McCreary, D., and Webb, S. (2004). Analysis of state exposure control to prevent cheating in online games. In *Proceedings of the 14th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV '04, pages 140–145, New York, NY, USA. ACM.

[LibVMI 2015] LibVMI (2015). Introduction to libvmi. https://tinyurl.com/y8d4xbq9.

[Liu et al. 2014] Liu, K., Lu, S., and Liu, C. (2014). Poster: Fingerprinting the publicly available sandboxes. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1469–1471, New York, NY, USA. ACM.

[Mandt et al. 2016] Mandt, T., Solnik, M., and Wang, D. (2016). Demystifying the secure enclave processor.

[Martignoni et al. 2010] Martignoni, L., Fattori, A., Paleari, R., and Cavallaro, L. (2010). Live and trustworthy forensic analysis of commodity production syst. In *Proc. 13th Intl. Conf. on Recent Advances in Intrusion Detection*, RAID'10. Springer-Verlag.

[Martignoni et al. 2009] Martignoni, L., Paleari, R., Roglia, G. F., and Bruschi, D. (2009). Testing cpu emulators. In *Proc. 18th Intl Symp. on Software Testing and Analysis*, ISSTA '09. ACM.

[Moon et al. 2012] Moon, H., Lee, H., Lee, J., Kim, K., Paek, Y., and Kang, B. B. (2012). Vigilare: Toward snoop-based kernel integrity monitor. In *Proc. 2012 ACM Conf. on Comp. and Comm. Sec.*, CCS '12. ACM.

[More and Tapaswi 2014] More, A. and Tapaswi, S. (2014). Virtual machine introspection: towards bridging the semantic gap. *Journal of Cloud Computing*, 3(1).

[Myers and Youndt 2007] Myers, M. and Youndt, S. (2007). An introduction to hardware-assisted virtual machine (hvm) rootkits. `https://tinyurl.com/y8wfsye5`.

[Neugschwandtner et al. 2010] Neugschwandtner, M., Platzer, C., Comparetti, P., and Bayer, U. (2010). danubis - dynamic device driver analysis based on virtual machine introspection. In Kreibich, C. and Jahnke, M., editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 6201 of *Lecture Notes in Comp. Science*. Springer Berlin Heidelberg.

[Nguyen et al. 2009] Nguyen, A. M., Schear, N., Jung, H., Godiyal, A., King, S. T., and Nguyen, H. D. (2009). Mavmm: Lightweight and purpose built vmm for malware analysis. In *Proc. 2009 Annual Comp. Sec. Applications Conf.*, ACSAC '09. IEEE Comp. Society.

[Opsahl 2013] Opsahl, J. M. G. (2013). *Open-source virtualization : Functionality and performance of Qemu/KVM, Xen, Libvirt and VirtualBox*. PhD thesis, Oslo Univ.

[Paleari 2015] Paleari, R. (2015). Fast coverage analysis for binary applications. `https://tinyurl.com/y7obk3y5`.

[Paleari et al. 2009] Paleari, R., Martignoni, L., Roglia, G. F., and Bruschi, D. (2009). A fistful of redpills: How to automatically generate procedures to detect cpu emulators. In *Proc. 3rd USENIX Conf. on Offensive Technologies*, WOOT'09. USENIX Association.

[Pappas et al. 2013] Pappas, V., Polychronakis, M., and Keromytis, A. D. (2013). Transparent rop exploit mitigation using indirect branch tracing. In *Proc. 22Nd USENIX Conf. on Sec.*, SEC'13. USENIX Association.

[Petroni et al. 2004] Petroni, Jr., N. L., Fraser, T., Molina, J., and Arbaugh, W. A. (2004). Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proc. 13th Conf. on USENIX Sec. Symp. - Volume 13*, SSYM'04. USENIX Association.

[Project 2015] Project, X. (2015). vmcs.c. `http://xenbits.xen.org/gitweb/?p=xen.git;a=blob;f=xen/arch/x86/hvm/vmx/vmcs.c`.

[Quynh and Suzaki 2010] Quynh, N. A. and Suzaki, K. (2010). Virt-ice: Next-generation debugger for malware analysis. `https://tinyurl.com/ybszcbxn`.

[Reina et al. 2012] Reina, A., Fattori, A., Pagani, F., Cavallaro, L., and Bruschi, D. (2012). When hardware meets software: A bulletproof solution to forensic memory acquisition. In *Proc. 28th Annual Comp. Sec. Applications Conf.*, ACSAC '12. ACM.

[Roemer et al. 2012] Roemer, R., Buchanan, E., Shacham, H., and Savage, S. (2012). Return-oriented programming: Syst., languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1).

[Rossow et al. 2013] Rossow, C., Dietrich, C., and Bos, H. (2013). Large-scale analysis of malware downloaders. In *Proc. of the 9th Inter. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'12. Springer.

[Rutkowska 2006] Rutkowska (2006). Subverting vista kernel for fun and for profit. `https://tinyurl.com/y86ltylh`.

[Rutkowska 2010] Rutkowska (2010). Qubes os project. `https://www.qubes-os.org/`.

[Rutkowska and Wojtczuk 2008] Rutkowska, J. and Wojtczuk, R. (2008). Preventing and detecting xen hypervisor subversions. `https://tinyurl.com/44denv2`.

[Schiffman and Kaplan 2014] Schiffman, J. and Kaplan, D. (2014). The smm rootkit revisited: Fun with usb. In *Availability, Reliability and Sec. (ARES), 2014 9th Intl. Conf. on*. IEEE.

[Schwarz et al. 2017] Schwarz, M., Weiser, S., Gruss, D., Maurice, C., and Mangard, S. (2017). Malware guard extension: Using sgx to conceal cache attacks. `https://arxiv.org/abs/1702.08719`.

[SeaBIOS 2015] SeaBIOS (2015). Seabios. `http://www.seabios.org/SeaBIOS`.

[Seshadri et al. 2007] Seshadri, A., Luk, M., Qu, N., and Perrig, A. (2007). Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proc. 21st ACM SIGOPS Symp. on Operating Syst. Principles*, SOSP '07. ACM.

[Shi et al. 2014] Shi, H., Alwabel, A., and Mirkovic, J. (2014). Cardinal pill testing of system virtual machines. In *23rd USENIX Sec. Symp. (USENIX Sec. 14)*. USENIX Association.

[Shinagawa et al. 2009] Shinagawa, T., Eiraku, H., Tanimoto, K., Omote, K., Hasegawa, S., Horie, T., Hirano, M., Kourai, K., Oyama, Y., Kawai, E., Kono, K., Chiba, S., Shinjo, Y., and Kato, K. (2009). Bitvisor: A thin hypervisor for enforcing i/o device sec. In *Proc. ACM SIGPLAN/SIGOPS Intl. Conf. on Virtual Execution Environments*, VEE '09.

[Sikorski and Honig 2012] Sikorski, M. and Honig, A. (2012). *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, San Francisco, CA, USA, 1st edition.

[Song et al. 2008] Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M. G., Liang, Z., Newsome, J., Poosankam, P., and Saxena, P. (2008). Bitblaze: A new approach to comp. sec. via binary analysis. In *Proc. 4th Intl. Conf. on Information Syst. Sec.*, ICISS '08. Springer-Verlag.

[Thober et al. 2008] Thober, M., Pendergrass, J. A., and McDonell, C. D. (2008). Improving coherency of runtime integrity measurement. In *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing*, STC '08, pages 51–60, New York, NY, USA. ACM.

[van Prooijen 2016] van Prooijen, J. (2016). The design of malware on modern hardware. `https://tinyurl.com/y8rwfj5t`.

[Vasudevan and Yerraballi 2006a] Vasudevan, A. and Yerraballi, R. (2006a). Cobra: Fine-grained malware analysis using stealth localized-executions. In *Proc. 2006 IEEE Symp. on Sec. and Priv.*, SP '06. IEEE Comp. Society.

[Vasudevan and Yerraballi 2006b] Vasudevan, A. and Yerraballi, R. (2006b). Spike: Engineering malware analysis tools using unobtrusive binary-instrumentation. In *Proc. 29th Australasian Comp. Science Conf. - Volume 48*, ACSC '06. Australian Comp. Society, Inc.

[Wang et al. 2010] Wang, J., Stavrou, A., and Ghosh, A. (2010). Hypercheck: A hardware-assisted integrity monitor. In *Proc. 13th Intl. Conf. on Recent Advances in Intrusion Detection*, RAID'10. Springer-Verlag.

[Wang et al. 2011] Wang, J., Zhang, F., Sun, K., and Stavrou, A. (2011). Firmware-assisted memory acquisition and analysis tools for digital forensics. In *Proc. 2011 6th IEEE Intl. Wksp on Systematic Approaches to Digital Forensic Engineering*, SADFE '11. IEEE Comp. Society.

[Wecherowski 2009] Wecherowski, F. (2009). A real smm rootkit: Reversing and hooking bios smi handlers. `https://tinyurl.com/knoms4t`.

[Will et al. 2017] Will, N. C., Condé, R. C. R., and Maziero, C. A. (2017). Mecanismos de segurança baseados em hardware: uma introdução à arquitetura intel sgx. `https://sbseg2017.redes.unb.br/wp-content/uploads/2017/04/20171107-SBSeg2017-Livro_de_Minicursos.pdf`.

[Willems et al. 2012a] Willems, C., Hund, R., Fobian, A., Felsch, D., Holz, T., and Vasudevan, A. (2012a). Down to the bare metal: Using processor features for binary analysis. In *Proc. of the 28th Annual Comp. Sec. Applications Conf.*, ACSAC '12. ACM.

[Willems et al. 2012b] Willems, C., Hund, R., and Holz, T. (2012b). Cxpinspector: Hypervisor-based, hardware-assisted system monitoring. Technical report, Horst Gortz Institute for IT Sec.

[Xia et al. 2012] Xia, Y., Liu, Y., Chen, H., and Zang, B. (2012). Cfimon: Detecting violation of control flow integrity using performance counters. In *Proc. 2012 42nd Annual IEEE/IFIP Intl. Conf. on Depend. Syst. and Net. (DSN)*, DSN '12. IEEE Comp. Society.

[Xu et al. 2017] Xu, J., Mu, D., Xing, X., Liu, P., Chen, P., and Mao, B. (2017). Postmortem program analysis with hardware-enhanced post-crash artifacts. In *26th USENIX Sec. Symposium*. USENIX.

[Yan et al. 2012] Yan, L.-K., Jayachandra, M., Zhang, M., and Yin, H. (2012). V2e: Combining hardware virtualization and softwareemulation for transparent and extensible malware analysis. In *Proc. 8th ACM SIGPLAN/SIGOPS Conf. on Virtual Execution Environments*, VEE '12.

[Zhang 2013] Zhang, F. (2013). Iocheck: A framework to enhance the security of i/o devices at runtime. In *2013 43rd Annual IEEE/IFIP Conf. on Depend. Syst. and Net. Wksp (DSN-W)*.

[Zhang et al. 2015] Zhang, F., Leach, K., Stavrou, A., Wang, H., and Sun, K. (2015). Using hardware features for increased debugging transparency. In *2015 IEEE Symp. on Sec. and Priv.* IEEE.

[Zhang et al. 2013] Zhang, F., Leach, K., Sun, K., and Stavrou, A. (2013). Spectre: A depend. introspection framework via system management mode. In *Proc. 43rd Annual IEEE/IFIP Intl. Conf. on Depend. Syst. and Net. (DSN)*, DSN '13. IEEE Comp. Society.