# QUEM VIGIA OS VIGILANTES? INTROSPECÇÃO DE MÁQUINAS VIRTUAIS PARA MONITORAMENTO DE SISTEMAS RTOS E GPOS EM PROCESSADORES ARM.

Fernando C. Menardi[1], William Z. Gomes[2]

1 Graduando em Engenharia de Controle e Automação, Câmpus São João da Boa Vista, fernando.caro@aluno.ifsp.edu.br
2 Professor de Engenharia de Controle e Automação, Câmpus São João da Boa Vista.

**RESUMO:** *Virtual Machine Instrospection* (VMI) é uma técnica que utiliza conceitos de virtualização para inspecionar o comportamento de uma aplicação ou sistema operacional em execução. Trata-se de uma abordagem de monitoramento furtiva que auxilia o monitoramento de agentes maliciosos, como *malwares*, que se encontram no estado da arte. O presente estudo busca apresentar uma abordagem para implementação de VMI em sistemas de arquitetura ARM através de um *Hypervisor Type-1* que contemple a virtualização de sistemas sistemas operacionais de propósito geral (GPOS) e sistemas operacionais de tempo-real (RTOS).

**PALAVRAS-CHAVE**: *Hypervisor*. *Virtual Machine Introspection*. VMI. *Malware*. ARM. RTOS.

## WHO WATCHES THE WATCHMEN? VIRTUAL MACHINE INTROSPECTION FOR RTOS AND GPOS ON ARM PROCESSORS

**ABSTRACT:** Virtual Machine Instrospection (VMI) is a technique that uses virtualization concepts to inspect the behavior of a running application or operating system. It is a stealth monitoring approach that helps to monitor state-of-the-art threats, such as malwares. The present study approaches the implementation of VMI in ARM architecture systems through a Type-1 Hypervisor that includes the virtualization of general purpose operating systems (GPOS) and real-time operating systems (RTOS).

**KEYWORDS**: Hypervisor. Virtual Machine Introspection. VMI. Malware. ARM. RTOS.

## INTRODUÇÃO

Os *malwares* que se encontram no estado da arte utilizam técnicas furtivas sofisticadas para modificar o comportamento do Sistema Operacional (SO) com o objetivo de ofuscar sua existência, bem como sua execução. É comum que através de arquiteturas de implementação de malware conhecidas, como *Rootkits*, malwares também consigam subverter mecanismos de segurança presentes no SO, garantindo que sua atividade permaneça indetectável. Portanto, é factível assumir que qualquer componente que é amparado pelo SO, em qualquer nível, incluindo componentes de segurança, tem sua credibilidade questionada quando se trata de ataques sofisticados.

Dentro do escopo da defesa, a virtualização oferece uma considerável redução dos vetores de ataque possíveis, e portanto, auxilia os agentes de defesa a vencer a corrida armamentista substancial contra agentes maliciosos, através da implementação de arquiteturas capazes de oferecer resiliência de operação e segurança. Para se beneficiar deste cenário, componentes de segurança também podem ser projetados utilizando conceitos de virtualização. Tal abordagem recebe o nome de *Virtual Machine Instrospection* (VMI) e está fundamentada na implementação de *Hypervisors*, softwares responsáveis por gerenciar Máquinas Virtuais (VM). O conceito de VMI estende a funcionalidade de um *Hypervisor* tornando possível observar, analisar e controlar o estado das Máquinas Virtuais de maneira furtiva e automatizada.

Por muito tempo a arquitetura x86 foi dominante nas implementações de sistemas computacionais de alto desempenho, entretanto, existe um significativo movimento de adesão e demanda da arquitetura ARM para este fim. Ambas as arquiteturas mencionadas possuem extensões de virtualização que são requisitos para a implementação de *Hypervisors* e utilização de VMI, contudo, a arquitetura ARM costuma entregar eficiência energética superior a x86, o que faz dela excelente candidata de aplicações embarcadas ou de alto desempenho.

É buscando corroborar com a crescente adesão dos sistemas baseados em ARM e com a necessidade de oferecer soluções de segurança sofisticadas, que o presente trabalho busca verificar as implicações de VMI para monitoramento de sistemas operacionais de tempo-real (RTOS) em conciliação com sistemas operacionais de propósito geral (GPOS), visto que o primeiro em especial, é de grande interesse para aplicações industriais. Para tanto, é necessário: mapear os vetores de implementação possíveis para a arquitetura ARM, que contemplem os conceitos de VMI; Caracterizar um *Hypervisor* compatível com sistemas RTOS e GPOS; Parametrizar as funcionalidades necessárias para que conceitos de VMI contemplem os interesses de monitorar, intervir e analisar a execução de atividades maliciosa.

Em suma, as contribuições deste artigo são:

1. Caracterização dos recursos dos processadores ARM para implementação de VMI.
2. Enumeração das técnicas de VMI pertinentes ao combate de ameaças que se encontram no estado da arte.
3. Implementação de um ambiente de desenvolvimento para aplicações de VMI em ARM.

## 2 FUNDAMENTAÇÃO TEÓRICA

O conceito de VMI é respaldado em vários mecanismos de virtualização que foram implementados e refinados ao longo das atualizações das arquiteturas dos processadores. É importante enfatizar que cada arquitetura possui extensões de virtualização distintas, isto é, os conceitos de virtualização são análogos, mas sua implementação é diferente. Além disso, mesmo considerando uma mesma arquitetura, neste caso a arquitetura ARM, detalhes da implementação podem mudar em versões diferentes. A seguir são apresentados os conceitos essenciais para o entendimento e discussão dos temas correlatos à virtualização, bem como oferecer insumo para a discussão dos tópicos de VMI.

### 2.1 Processadores ARM

ARM é a sigla para *Advanced RISC Machine***.** Como já declarado pelo nome, os processadores ARM são do tipo *Reduced Instruction Set Computer* (RISC). O propósito da arquitetura RISC é oferecer um simples e poderoso conjunto de instruções capazes de executar dentro de um único ciclo de clock[1], em uma alta velocidade. Essa abordagem é contrária à adotada por outras famílias de arquitetura tradicionais como a x86[2], que é do tipo *Complex Instruction Set Architecture* (CISC), onde o conjunto de instruções contempla operações complexas, mas que consomem mais ciclos de clock para executar (SLOSS, 2004).

Os processadores ARM foram idealizados com a finalidade de integrar sistemas embarcados, portanto foram pensados para serem de pequenas dimensões, baixo consumo de energia e baixo consumo de memória, se comparado a outros processadores. Essas características com o tempo se tornaram atrativas para outras aplicações como: celulares, servidores e computadores pessoais (SLOSS, 2004).

---

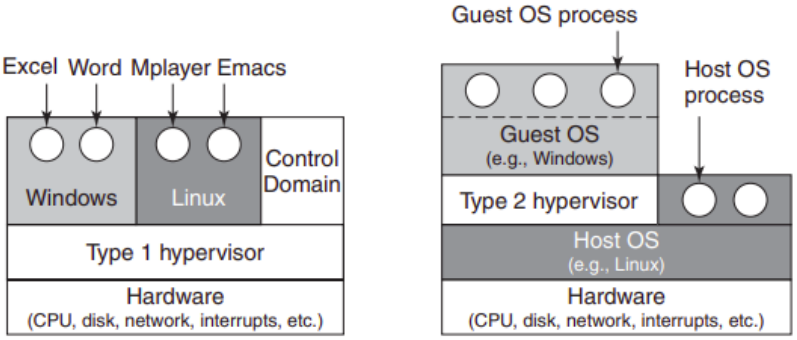
[1] Sinal periódico de grande precisão, normalmente na faixa de centenas de megahertz a alguns gigahertz (TANENBAUM, 2015).

[2] Processadores modernos baseados na família de arquiteturas de conjuntos de instruções que começaram com o Intel 8086 na década de 1970 (TANENBAUM, 2015).

## 2.2 *Hypervisors*

O nome *Hypervisor* é utilizado para designar softwares destinados à virtualização de sistemas operacionais. Existem duas abordagens distintas de virtualização, o que implica na existência de dois tipos distintos de *Hypervisor*, os chamados "*type 1*" e "*type 2*". Os *hypervisors type 1* rodam direto no hardware, como um sistema operacional tradicional, sendo o único programa que roda com o maior nível de privilégio possível. Em contrapartida, os *hypervisors type 2* são programas que dependem de um sistema operacional (Linux ou Windows, por exemplo), para executar. A Figura 1 apresenta a diferença de arquitetura entre a implementação de um *Hypervisor type 1* e *type 2* (TANENBAUM, 2015).

Figura 1 - ARM Exception Levels




Fonte: TANENBAUM, 2015

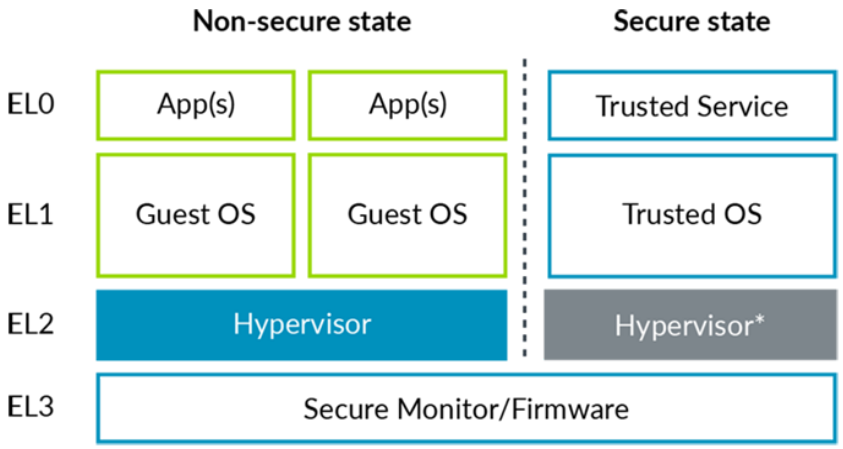
A partir da Figura 1 fica evidente que o *Hypervisor type 2* introduz uma camada de abstração adicional se comparado com o *type 1*, o que impacta a performance diretamente. Portanto, apesar de ambos serem capazes de virtualizar sistemas, se a intenção é que o sistema virtualizado tenha desempenho muito próximo de uma execução nativa, isto é, como se não estivesse sendo virtualizado, a opção adequada é o *Hypervisor type 1* (TANENBAUM, 2015).

Como virtualização é uma funcionalidade bastante explorada comercialmente nas arquiteturas de sistemas modernos, as arquiteturas dos processadores ganharam extensões (Intel VT-d/VT-x, AMD-V, ARM VE e etc) destinadas a aceleração de mecanismos de virtualização, que tem como objetivo aproximar ainda mais o desempenho de um sistema virtualizado a um sistema nativo (TANENBAUM, 2015).

## 2.3 ARM *Exception Levels*

Na arquitetura ARM a execução de instruções pode ocorrer em diferentes níveis de privilégio. Esses níveis são chamados de *Exception Levels* (ELn, onde "n" indica o nível). A partir da versão ARMv8-A, também chamada de AArch64 ou ARM64, são implementados quatro *Exception Levels*, do EL0 ao EL3. Na Figura 2, é apresentada a hierarquia desses níveis, sendo que o privilégio de execução aumenta de acordo com o aumento do nível (ARM, 2019b).

Figura 2 - ARM Exception Levels




Fonte: ARM, c2019b

Tradicionalmente, os níveis EL1 e EL0 são utilizados para a execução de componentes do SO (*kernel*[3] e aplicações *user mode*[4], respectivamente); o nível EL2 é destinado a execução do *Hypervisor* (quando implementado) para virtualização de sistemas operacionais; o nível EL3 é utilizado para a execução do firmware[5], tipicamente durante a fase de inicialização (*boot*) do sistema. Tratando-se de uma arquitetura que pressupõe virtualização, o software que é executado em EL2 tem acesso aos mecanismos de controle de virtualização: *Second Level Address Translation*[4] (SLAT)**;** controle sobre *trapping*[6], na execução de instruções e acesso a registradores específicos; geração de *virtual exceptions*[7] (ARM, 2019b).

Ainda na Figura 1, é notável que a partir de EL2 é estabelecida uma divisão entre *non-secure state* e *secure state*, estes são modos de execução estabelecidos pelo módulo ARM *TrustZone,* uma extensão dos processadores ARM destinada ao isolamento de componentes críticos de segurança. O objetivo é permitir que um sistema executando em *secure state*, também conhecido como *Trusted Execution Environment* (TEE ou *Secure World*), ofereça funcionalidades que podem ser requisitadas por um sistema executando em *non-secure state*, também conhecido como *Rich Execution Environment* (REE ou *Insecure World*), sem que o REE tenha acesso ao contexto de execução do TEE. Portanto, softwares executados em estados diferentes, têm também uma visão diferente do sistema, dessa forma, dispositivos, funcionalidades de segurança ou até mesmo credenciais, podem ser mantidas escondidas de softwares não-críticos que estejam rodando no REE. É possível entender essa divisão como sendo semelhante ao tradicional sistema de *ring protection* entre *user mode* e *kernel mode*, típica dos processadores x86, mas considerando que neste caso, tal divisão não se limita somente a CPU, ela afeta também os barramentos do sistema para dispositivos periféricos e controladores. A transição entre TEE e REE é realizada por funções mantidas em EL3, sendo esta restrição reforçada a nível de hardware (ARM, 2019d).

---

[3] Software do núcleo do Sistema Operacional, com maior nível de privilégio, possuindo acesso direto ao hardware (TANENBAUM, 2015).

[4] Software que é executado sob um Sistema Operacional, fora do núcleo (kernel). Em geral, possui baixo nível de privilégio e não tem controle sobre o hardware (TANENBAUM, 2015).

[5] Software que persiste na memória não volátil e é raramente atualizado. Sua funcionalidade típica é atuar na inicialização do dispositivo, antes da execução de um sistema operacional (TANENBAUM, 2015).

[6] Execução de instruções que mudam o modo de operação do processador para um nível de privilégio maior., pois a instrução não pode ser recomeçada (TANENBAUM, 2015).

[7] Situação anômala ou excepcional que exige processamento especial, em geral, com maiores níveis de privilégio do que o atual. Em geral, causa a preempção da rotina em execução e transfere a execução para um código dedicado à lidar com esse tipo de ocorrência (TANENBAUM, 2015).

Processadores ARM sempre iniciam a execução no maior *exception level* implementado (tipicamente EL3). Em um fluxo de execução típico a execução inicia em EL3 (onde o firmware executa rotinas de inicialização do hardware e periféricos) e assim que possível, transfere a execução para o SO em EL1. No entanto, no caso de interesse, onde os sistemas operacionais distintos vão executar paralelamente, sobre um hypervisor, é necessário mudar o nível para EL2, inicializar o *Hypervisor*, para só então passar para EL1 (ARM, 2019b).

Quando ocorre uma *exception*, o processador precisa executar uma função, também chamada de *handler*, que corresponde ao tipo de *exception* que ocorreu. Os diferentes *handlers* para os diferentes tipos possíveis de *exception*, apresentados na Figura 3, são alocados e armazenados em memória durante a inicialização do sistema. A localização em memória onde um *handler* está localizado se chama *Exception Vector*. Os *handlers* são armazenados sequencialmente, em uma tabela, em uma ordem pré-definida pela arquitetura. Esta tabela é chamada de *Exception Vector Table* (ARM, 2019c).

Figura 3 - Modelo ARM *Exception Vector Table*


Fonte: (ARM, 2019c)

Existe uma *Exception Vector Table* independente para EL3, EL2 e EL1. O endereço base em memória das tabelas é armazenado nos registradores “VBAR_ELn”, onde “ELn” é o *exception level* correspondente (ARM, 2019c).

Na arquitetura ARM não há maneiras de um código em execução aumentar seu próprio *exception level* sem a participação de algum código que já executa em um nível superior. Este formato, garante que nenhum programa seja capaz de escapar do *exception level* que lhe foi atribuído, em outras palavras, a única maneira de se alterar a *exception level* é gerando uma *exception*, que será tratada por um e*xception level* superior, que pode ser decorrente da (ARM, 2019c):

1. Execução de uma instrução ilegal (tentativa de acessar um endereço de memória que não existe, por exemplo).
2. Execução de uma instrução “svc” ou “hvc”, utilizada para gerar um *exception* propositalmente.
3. Interrupção de Hardware (interrupções geradas por hardware são tratadas como um tipo especial de *exception*).

Dentro do contexto de virtualização, entender o papel das *exceptions* é fundamental para compreender como funcionam as denominadas *Hypercalls*. Uma *Hypercall* é um tipo especial de *exception*, chamada de *Hypervisor Call Exception* que resulta na transição para EL2. Uma *Hypercall*

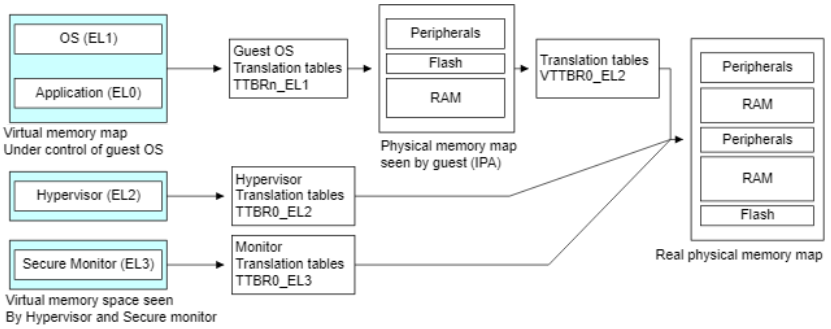
para um *Hypervisor* possui a mesma função de uma *system call*[8] (*syscall*) para um OS. Essa funcionalidade é o meio pelo qual uma VM pode requisitar que uma operação privilegiada seja executada pelo *Hypervisor*, como por exemplo, atualizar as *pagetables*[9], criar uma nova VM e etc. Quando uma *Hypercall* é feita e a transição para EL2 acontece, é executado o código presente no *offset el1_sync* (0x400, na Figura 2) da *Exception Vector Table*, cujo endereço está armazenado no registrador *VBAR_EL2* (ARM, 2019c).

## 2.4 *Second Level Address Translation* (SLAT)

Toda vez que uma aplicação acessa uma região de memória os endereços são traduzidos pelo módulo MMU (*Memory Management Unit*), atrelado a CPU, utilizando tabelas programadas e armazenadas em memória pelo *Kernel* ou *Hypervisor*. Essas tabelas também especificam propriedades referentes à permissões de acesso e de execução referentes a regiões específicas (ARM, 2019a).

O suporte de virtualização do ARM estende o funcionamento tradicional da MMU através de um segundo estágio (opcional) de tradução de endereços. O *Second Level Address Translation* (SLAT) é uma tecnologia de virtualização assistida por hardware que permite que o Hypervisor tenha controle sobre quais segmentos da memória uma VM (máquina virtual) pode acessar e onde esses recursos aparecem no espaço de memória visto na perspectiva da VM. Este recurso é essencial para o conceito de virtualização, pois garante que uma VM seja capaz de ver ou acessar somente os recursos que lhe foram atribuídos, não conseguindo acessar ou ver os recursos alocados para outras VMs ou para o Hypervisor em si (ARM, 2019a).

Figura 4 - Processo de tradução dos endereços de memória utilizando SLAT.




Fonte: ARM, 2019a

Na Figura 4 é apresentado um diagrama simplificado do SLAT na arquitetura ARM. Em sistemas tradicionais que não contemplam virtualização, de maneira geral, apenas um estágio de tradução é implementado através da MMU. É evidente que a adição de um estágio subsequente acarreta uma perda de performance, entretanto, as extensões de virtualização oferecem módulos de MMU preparados para lidar com este processo a nível de hardware, o que acelera esse processo (ARM, 2019a).

Como já mencionado, o processo de tradução necessita que as tabelas de tradução de endereços sejam armazenadas em memória. O endereço das tabelas do primeiro e do segundo estágio são armazenadas respectivamente nos registradores "TTBRn" e "VTTBR/VSTTBR" (VSTTBR armazena entradas exclusivas para traduções envolvendo o *Secure World*). O espaço de memória de

[8] Execução de uma instrução capaz de requisitar serviços do sistema operacional, que em geral só podem ser executados no kernel (TANENBAUM, 2015).

[9] Estrutura de dados usado pelo sistema de memória virtual em um sistema operacional. Sua função é mapear endereços virtuais (*virtual page number*) para endereços físicos (*physical frame number*) (TANENBAUM, 2015).

EL1 e EL0 é dividido entre dois "TTBRn", sendo "TTBR0_EL1" e "TTBR1_EL1". Esta divisão é apenas uma forma conveniente de seguir a divisão tradicional do espaço de memória de sistemas operacionais que venham ser virtualizados, e que por sua vez, possuem seu espaço de memória também dividido entre o *kernel* e para as aplicações em *usermode* (ARM, 2019a).

No primeiro estágio, um Endereço Virtual (VA) é traduzido para um Endereço Físico Intermediário (IPA), processo este, que usualmente está sob controle do SO. O segundo estágio, controlado pelo *Hypervisor*, é então responsável por realizar a tradução final entre o IPA e o Endereço Físico em memória (PA). Para tornar esse sistema de traduções eficiente utiliza-se entradas de TLBs[10] (*translation lookaside buffer*), que funcionam como um *cache*[11] para traduções recorrentes (ARM, 2019a).

### 2.5 ARM *Translation Lookaside Buffers*

A função de uma TLB (*translation lookaside buffer)* é simples: armazenar em cache o resultado do mapeamento entre memória virtual e física (incluindo os casos em que existe SLAT), para que o processo de "tradução" não seja necessário para segmentos de memória os quais o acesso é recorrente. A implementação de uma TLB evita sobrecargas do sistema de tradução e consequentemente permite um ganho de *performance*.

Na arquitetura ARM cada VM (ou Dominio, na terminologia do Xen) recebe um *Virtual Machine Identifier* (VMID) que é utilizado para rotular entradas na TLB. O objetivo é identificar a qual VM, cada entrada pertence. Esse sistema de rótulos, permite que traduções para múltiplas VMs estejam presente nas TLBs ao mesmo tempo. Um detalhe importante é que o VMID é associado ao registrador "VTTBR", portanto, a CPU não precisa realizar a limpeza (*flush*) da TLB durante qualquer *context switch*, garantindo a *performance* geral do sistema. (ARM, 2019a).

A organização da TLB é dividida em dois conjuntos separados: TLB de instrução (iTLB) e TLB de dados (dTLB). O iTLB armazena em *cache* as traduções referentes a buscas de instruções, enquanto que o dTLB armazena traduções referentes às buscas de dados.

### 2.6 XEN *Hypervisor*

Xen é um software livre de virtualização que contempla várias arquiteturas adotadas pelo mercado, incluindo a arquitetura ARM. Tratando-se de um *Hypervisor type 1*, o mesmo é executado somente e exclusivamente em EL2, deixando EL1 para o kernel do SO virtualizado (*guest kernel*) e EL0 para as aplicações em user mode (*guest user space*) (XEN, 2018b).

Na terminologia do projeto Xen, as VMs são chamadas de "*Unprivileged Domains*", abreviadamente, apenas DomU. Quando o Xen é inicializado a primeira VM é criada e esta recebe o nome especial de "*Dom0*". Este é um domínio privilegiado, que executa um SO adaptado (paravirtualizado), no qual são implementadas e é permitido realizar chamadas de controle (*hypercall*) para o *Hypervisor*. Em outras palavras, o Dom0 é a instância virtualizada pelo Xen, responsável por implementar as interfaces de controle do *Hypervisor*. Os demais domínios (DomU, sigla para *Unprivileged Domain*), não possuem privilégio para realizar *hypercalls*, e a princípio, não possuem mecanismos implementados para realizar as mesmas. Na Figura 5 é apresentada a arquitetura simplificada adotada pelo Xen para implantação do sistema de virtualização, onde o Xen

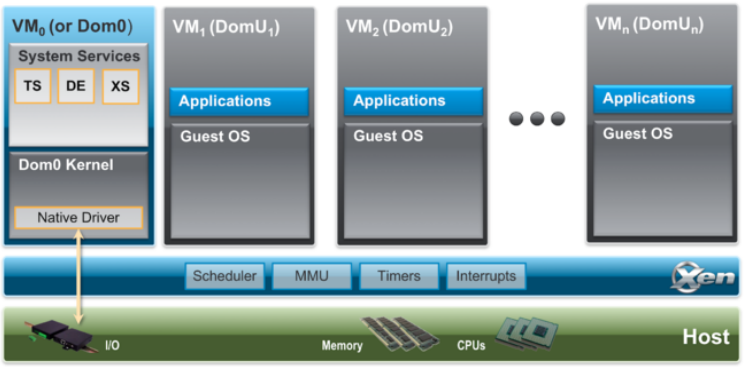
---

[10] Dispositivo de hardware utilizado para mapear endereços virtuais para endereços físicos sem a consulta de uma *pagetable*. É dividido em entradas que contém informações sobre uma página, incluindo o número da página virtual, um bit que é definido quando a página é modificada, a proteção da memória (permissões de leitura/gravação/execução) e o *frame* de página física no qual a página está localizada. Esses campos têm uma correspondência um-para-um com os campos da *pagetable*, exceto o número da página virtual, que não é necessário na *pagetable*. Outro bit indica se a entrada é válida. É geralmente integrado na MMU (TANENBAUM, 2015).

[11] Unidade de memória de acesso mais lento se comparada com um registrador, mas mais rápida se comparada com a memória principal do sistema (ram). Integrada ao processador. Quando o programa precisa acessar um conteúdo na memória, o hardware de cache verifica se a entrada necessária já está no cache. Se for o caso, a solicitação é satisfeita sem nenhuma solicitação de acesso à memória principal, o que implica em um acesso mais rápido (TANENBAUM, 2015).

em si, sendo um *hypervisor*, roda diretamente no hardware, enquanto todos os demais componentes do sistema rodam como VMs acima deles (XEN, 2018b).

Figura 5 -Arquitetura simplificada do XEN *Hypervisor*.




Fonte: XEN, 2018b

A partir desse diagrama, fica evidente que o *Hypervisor* adiciona uma camada de abstração para todos os sistemas operacionais virtualizados. É válido reforçar que somente o "Dom0" possui conhecimento e meios formais de reconhecer essa abstração, os demais domínios (DomU), não tem necessidade de reconhecer a existência de um *Hypervisor* (XEN, 2018b).

Partindo do princípio de que somente o *Hypervisor* tem acesso direto ao hardware é necessário que de alguma forma esse acesso ao hardware seja repassado para as VMs. O Xen utiliza SLAT na MMU para alocar memória para as máquinas virtuais (XEN, 2018a).

Cada *core*[12] (pCPU) existente na CPU do *host* (hardware), conectada ao Xen é abstraído para uma unidade chamada "vCPU" ou "*virtual CPU*". Os processos em execução, que nesse caso são as VMs em si, rodam de forma exclusiva na "vCPU" que lhe foi atribuída por um certo período de tempo até ser interrompido. Quando isso acontece, o estado da CPU (registradores) são salvos e outro processo passa a ser executado na CPU. Este processo é análogo ao processo de *context switch*[13] implementado em sistemas operacionais. O processo de interrupção e realocação dos processos no Xen é coordenado por um componente chamado *scheduler*, o papel dele é decidir dentre todas as vCPUs disponíveis e alocadas para várias VMs, qual deve rodar em cada pCPU, em um determinado periodo de tempo. O Xen pode ter seu *scheduler* configurado para utilizar algoritmos diferentes de acordo com o tipo de aplicação (XEN, 2019b).

Todo o hardware, incluindo os periféricos conectados ao host devem ser descritos em um arquivo do tipo *devicetree*, para que sejam localizados e reconhecidos pelo Xen no momento da inicialização. Este arquivo possui uma estrutura de dados de árvores com nós que descrevem os dispositivos no sistema. Cada nó possui um par de propriedades e valores que descrevem as características do dispositivo que está sendo representado, sendo essas características, especificações I/O, mapeamento de interrupções, GPIOs, clock. A organização da estrutura é de simples compreensão, cada nó tem exatamente um parente, exceto pelo nó raiz, que não tem parentes (DEVICETREE, 2021).

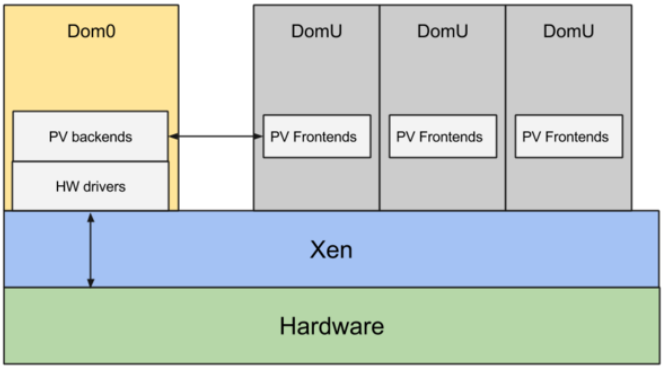
---

[12] Núcleo de processamento individual. Os processadores modernos possuem mais de uma unidade de processamento e são chamados de *multicore* (TANENBAUM, 2015).

[13] Em um sistema operacional capaz de executar vários processos, a operação de alternar de um processo para outro, é chamado de *context switch*. (TANENBAUM, 2015).

Tratando-se de periféricos gerais, como discos e placas de rede, o Xen permite que as VMs utilizem os mesmos através do Dom0. Os sistemas operacionais rodando como DomUs ganham acesso a um conjunto de dispositivos virtuais genéricos ao rodar os chamados *paravirtualized frontend drivers*, como mostrado na Figura 6 (XEN, 2018a).

Figura 6 - Diagrama de acesso de uma VM (DomU) aos dispositivos de hardware através do Dom0, utilizando drivers paravirtualizados.




Fonte: XEN, 2018a

Os *paravirtualized frontend drivers* (*PV Frontend*) se conectam aos *paravirtualized backends drivers* (*PV Backend*) que rodam no Dom0, sendo este, o domínio em que os dispositivos são mapeados utilizando *memory-mapped I/O*[14] (*MMIO*) . Essa conexão é realizada utilizando memória compartilhada e a notificação de eventos entre um PV Frontend e um PV Backend é realizada através de *software interrupts*[15] . A relação de existência relacional do PV Backend para PV Frontend é de "1:n", isto é, um único "*backend*" é capaz de suprir múltiplos "*frontends*" (XEN, 2018a).

## 2.7 ARM *Interrupts* e Xen *Events*

Nos processadores ARM as interrupções são gerenciadas pelo *Generic Interrupt Controller*. A arquitetura GICv3 e GICv4, presentes nas famílias ARMv8, auxiliam no processo de virtualização uma vez que possibilitam definir uma interface *virtual GIC* (vGIC) para cada CPU. Esta interface possibilita que o *Hypervisor* gerencie interrupções, possibilitando que interrupções sejam roteadas para as diferentes instâncias de VM em execução. Qualquer tentativa de configurar o GIC por uma VM causa o desvio do fluxo de execução para EL2, onde o *Hypervisor* consegue emular a operação (ARM, 2019c).

Na arquitetura do Xen, *interrupts* são referenciadas como *events*. As interrupções são mapeadas para os chamados *event channels*, e são entregues de forma assíncrona a VM (DomU) correspondente, usando funções de *callback* registradas através de uma *hypercall*. Os *events* são especialmente importantes para lidar com a intercomunicação entre VMs. Especialmente porque é através deles, que uma VM privilegiada (Dom0) pode registrar funções (*callbacks*) para serem executadas conforme um evento é gerado.

---

[14] Abordagem de acesso à dispositivos periféricos e controladores através do mapeamento dos seus respectivos registradores direto no espaço de endereços do processador.. Nessa abordagem, na perspectiva do processador, o acesso (seja de leitura ou escrita) dos registradores é realizado como um acesso usual à memória principal (TANENBAUM, 2015).

[15] Uma classe de *exception*. *Interrupts* podem ser geradas por hardware ou por software e são assíncronas. Quando ocorrem, interrompem o fluxo de execução e redireciona-o para a execução de uma função própria para lidar com o interrupções. É majoritariamente usada para preempção de processos em execução e comunicação de eventos lançados por dispositivos virtuais ou de hardware (TANENBAUM, 2015).

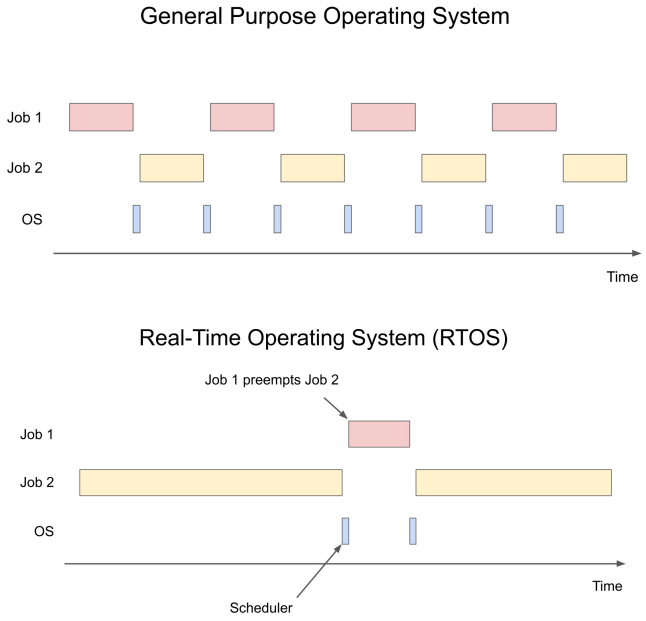
## 2.8 *Virtual Machine Introspection* (VMI)

Uma das grandes vantagens de se virtualizar um SO é que o *Hypervisor* utilizado tem acesso total ao contexto de execução do mesmo. Além disso, quando não é utilizado um SO "paravirtualizado", a dificuldade de se detectar que o SO está sendo virtualizado, pela perspectiva do SO em si, é elevada. Assim, softwares em execução no SO virtualizado (incluindo agentes maliciosos), têm a falsa impressão de estarem sendo executados em um ambiente nativo, e através do *Hypervisor*, é possível monitorar toda e qualquer atividade desempenhada pelos mesmos sem que esse monitoramento seja detectado. O objetivo das técnicas de VMI é instrumentalizar um *Hypervisor* para introspectar as VMs em execução, isto é, monitorar e analisar o contexto de execução.

## 2.9 Sistema Operacional de Tempo-Real

Um sistema operacional de tempo-real (RTOS) se difere de um sistema operacional de propósito geral (GPOS) no tempo de execução das tarefas. Em um RTOS, as tarefas possuem prazos bem definidos e críticos para serem executadas. Para que isso aconteça, os sistemas RTOS implementam *schedulers* com algoritmos específicos que contemplem esse objetivo. Na Figura 10 é apresentada uma comparação entre o comportamento dos *schedulers* de um GPOS e um RTOS, respectivamente (SIEWERT, 2016).

Figura 10 - Comparação do agendamento de tarefas de um GPOS e um RTOS.




Fonte: DIGI-KEY ELECTRONICS, 2022

Uma regra geral é que em um GPOS as tarefas recebem tempos iguais de execução na CPU, enquanto que em um RTOS, as tarefas recebem tempos de execução proporcionais à prioridade de execução definida. No caso de uma tarefa A de menor prioridade atrasar na execução, uma tarefa B de maior prioridade interrompe a execução da mesma e passa a executar em seu lugar, assim, é garantido que o prazo de execução da tarefa B é respeitado. Assim é possível considerar que um RTOS é determinístico, entretanto, entrega um bom desempenho geral, mas pode eventualmente sacrificar o desempenho para cumprir prazos e ser determinístico (SIEWERT, 2016).

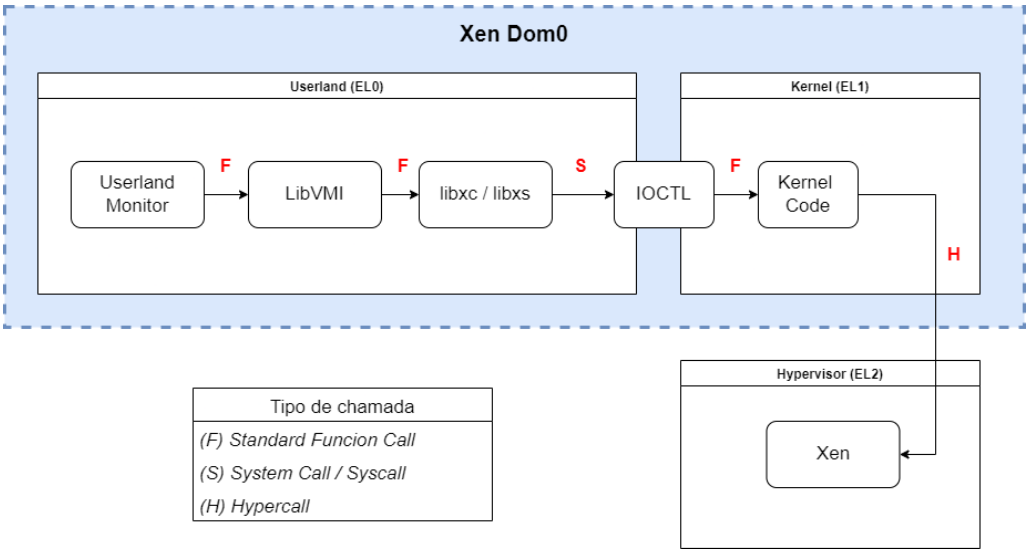
# 3 METODOLOGIA

Nesta seção é apresentada a arquitetura do sistema proposto e a metodologia escolhida para implementação, viabilizando as funcionalidades propostas.

## 3.1 Arquitetura

O objetivo desta pesquisa está completamente vinculado aos processadores ARM, mas limita-se às famílias "ARMv8", ou superiores, de arquitetura 64 bits, uma vez que estas possuem extensões de virtualização e possuem funcionalidades necessárias para implementação de recursos de VMI. O hypervisor escolhido como plataforma é o Xen, desenvolvido como parte do Xen Project. Ele contempla a virtualização de sistemas GPOS e RTOS e é suportado pelo "LibVMI", um importante *framework* de referência para implementação de recursos de VMI em diversos sistemas e arquiteturas. Na Figura 11 é apresentado como ocorre o fluxo de comunicação entre os diferentes componentes do sistema.

Figura 11 - Integração dos componentes do sistema de VMI utilizando LibVMI e Xen.




Fonte: Elaboração Própria

O LibVMI utiliza *shared objects*, também chamados de *shared libraries,* para executar funções do Xen. Esses *shared objects*, que funcionam como bibliotecas, são dependências instaladas junto ao Xen no sistema operacional utilizado como Dom0. Portanto, os componentes libxc e libxs são binários compiladores importados pelo LibVMI. Essas duas bibliotecas são utilizadas principalmente para requisitar a execução de funções do *kernel*, através de IOCTLs[16], um tipo especial de *syscall*, realizando a transição de EL0 para EL1. Essas funções presentes no kernel do Dom0 fazem a preparação e a chamada das *Hypercalls*. É somente depois disso, que ocorre a transição de EL1 para EL2, e o *Hypervisor* ("Xen Core", na Figura 11) é chamado para executar a operação.

## 3.2 Interceptação e Monitoramento

[16] Um tipo de *syscall* específica que realiza operações para dispositivos específicos. São uma alternativa para drivers implementarem suas próprias operações dentro do kernel

Uma das possíveis abordagens para interceptar a execução de funções do sistema, é através da remoção das permissões de acesso (remoção da permissão de execução, por exemplo) da região de memória onde se encontram as funções (processo este, beneficiado pelo SLAT). Isso faz com que os acessos causem *exceptions* forçando a transição de EL0 ou EL1 para EL2, permitindo que o *Hypervisor* assuma o controle. Entretanto, essa abordagem adiciona alguns problemas (PROSKURIN. 2018):

1. Acarreta uma considerável sobrecarga no sistema, uma vez que vários dos acessos à memória correspondem a execução de trechos de código irrelevantes para o monitoramento.
2. No caso do AArch32 (ARM 32 bits) o SLAT não suporta a funcionalidade de manter as permissões de acesso a memória como "somente execução" (*execute-only*), toda página de memória que contém código, deve ser possível de ler e executar pela CPU ou vCPU.

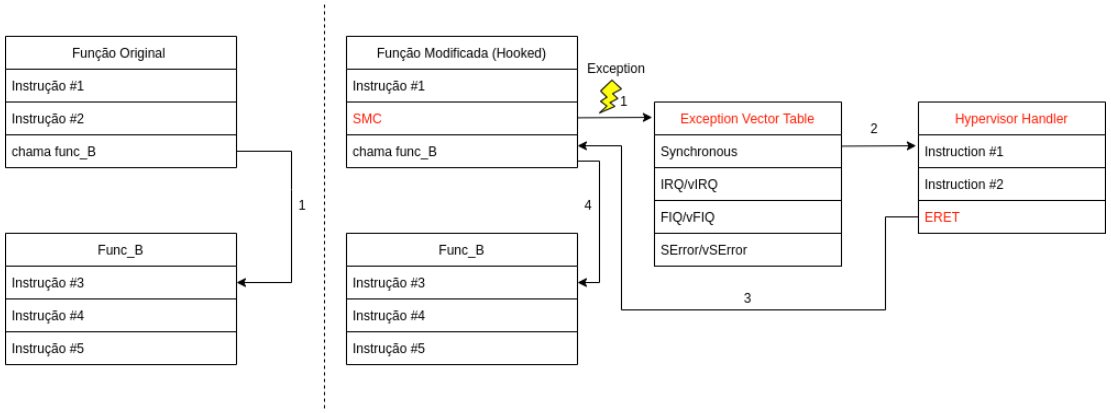
### 3.2.1 Interceptação de execução utilizando *Syscall Hooking*

Para superar os problemas apontados, uma das soluções é a utilização da técnica de *syscall hooking*, que neste caso, possui a finalidade de desviar o fluxo de execução para o contexto do *hypervisor* (EL2) somente quando uma função ou *syscall* de interesse seja executada. Implementar *syscall hooking* trata-se efetivamente de modificar instruções ou ponteiros carregados em memória, para desviar o fluxo de execução, e assim, conseguir executar funções adicionais.

Tratando-se de um ARM, para desviar o fluxo de execução para EL2, uma *exception* (*SIGTRAP*) pode ser gerada utilizando uma instrução "*SMC*" (a instrução que gera a *exception* também é chamada de "*trapping instruction*"). Desta forma, o *handler* registrado na *Exception Vector Table* será executado, como mostrado na Figura 8. O código deste *handler* pode ser personalizado para registrar os detalhes do contexto de execução no momento da chamada, concluindo a operação de monitoramento em si.

Outras instruções podem ser utilizadas para este fim, o que faz da instrução "*SMC*" uma alternativa adequada é que, ao contrário de instruções específicas de *breakpoint*, as *exceptions* geradas por execução de uma instrução "SMC" só podem ser direcionada para uma TEE no *TruztZone* ou para o *Hypervisor* (PROSKURIN, 2018).

Figura 12 - Desvio do fluxo de execução (*Hooking*) utilizando a instrução SMC na arquitetura ARM




Fonte: Elaboração Própria

É evidente que utilizar o *hooking* adiciona etapas adicionais à execução, sendo este o motivo da sobrecarga do sistema quando utilizada frequentemente. Entretanto, para interceptações pontuais, é plausível assumir que o aumento da carga de execução do sistema é mínima, e portanto, não oferece grandes perdas de desempenho.

No Xen, as *exceptions* são mapeadas para os *event channels*, que por sua vez são acessíveis pelo Dom0. Dessa forma a LibVMI utiliza os *event channels* para a implementação de *handlers*

dentro do próprio Dom0, visando não precisar realizar nenhuma modificação nos *handlers* originais do Xen, registrados durante o boot, na *Exception Vector Table*.

É factível considerar que um agente malicioso pode fazer varreduras na memória e detectar que algumas instruções foram modificadas intencionalmente para barrar suas operações. É necessário então que os acessos à memória sejam controlados e que seja possível "esconder" as modificações das instruções modificadas pelo agente de monitoramento. Uma solução possível é alterar as permissões de memória (abrindo mão da compatibilidade com AArch32), como já tratado anteriormente, desta forma, mediante à uma tentativa de leitura das instruções modificadas, é possível interromper a execução e retornar as instruções originais na memória antes que o agente malicioso consiga de fato concluir a operação de leitura. Entretanto isso adiciona mais um problema:

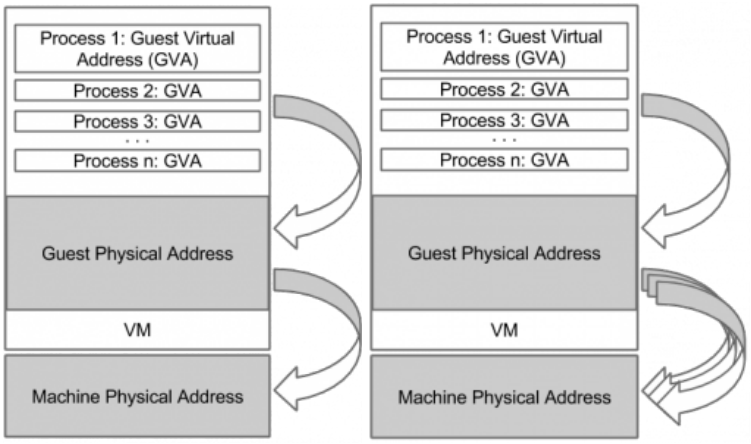
1. Modificar as permissões de acesso em tempo de execução pode ocasionar "*race conditions*" no monitoramento.

Considerando que mais de uma vCPU existe dentro do sistema: a remoção da instrução que ocasiona a *exception* proposital, de dentro da função alvo (por conta da tentativa de varredura executada por uma vCPU), pode possibilitar que uma segunda vCPU execute a função alvo sem que seja possível interceptar. Entretanto, através de um recurso chamado "altp2m", implementado pelo Xen, é possível mitigar essa *race condition*.

### 3.2.2 Xen "altp2m"

Na terminologia do Xen a camada de gerenciamento de memória é chamada de "p2m" (*physical memory to machine physical*) e é a responsável pela implementação do SLAT, que por sua vez, na é chamada de *Hardware Assisted Paging* (hap). Como ilustrado pela Figura 9, em implementações tradicionais, onde o SLAT é utilizado, o Hypervisor mantém uma *pagetable* adicional (além da *pagetable* já mantida pelo sistema operacional virtualizado) que contém as informações suficientes para lidar com as traduções. Tradicionalmente a *pagetable* é instanciada quando a VM é criada e não muda (LENGYEL, 2016).

Figura 9 - Processo de SLAT utilizando Xen: "p2m" à direita e "altp2m" à esquerda.




Fonte: LENGYEL, 2016

A implementação do módulo chamado "altp2m" permite que sejam instanciadas múltiplas *memory views (pagetables),* simultaneamente, como ilustrado na Figura 9, no processo apresentado à direita. Essa funcionalidade é importante para implementação de funcionalidades de monitoramento, pois sendo possível manter múltiplas *pagetables*, é possível manter cópias de uma *pagetable*, chamadas de *shadow copy* (LENGYEL, 2016).

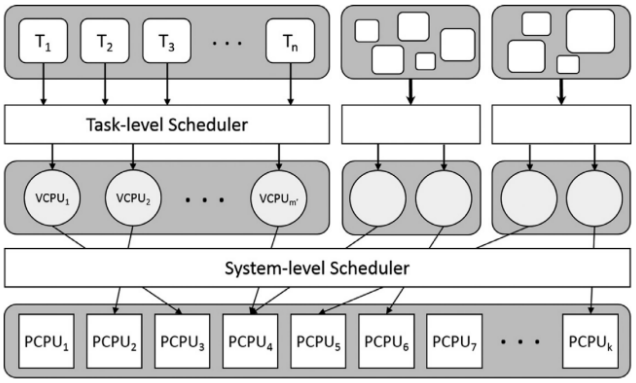
Ao invés de modificar as permissões em tempo de execução, é utilizado então o “altp2m” para instanciar *memory views* (*shadow copies*) com permissões diferentes e alterna-las em tempo de execução modificando o endereço do registrador “VTTBR” do “DomU” em supervisão. Além disso, *hypervisor* pode designar uma *memory view* específica para cada vCPU.

Essa granularidade no tratamento da memória é fundamental para mitigar o problema de *race conditions* mencionado anteriormente, pois permite que somente as permissões da memória atribuída especificamente à uma vCPU seja alterada (PROSKURIN, 2018).

### 3.3 Suporte à sistemas RTOS

O maior desafio quanto ao suporte de sistema RTOS, é o problema de aninhamento de *schedulers*. Como evidenciado na Figura 10, em um ambiente virtualizado o *scheduler* do *Hypervisor* (*System-level scheduler*) divide o tempo de execução para cada VM, entretanto, o mesmo é realizado pelos *schedulers* de cada sistema operacional (*Task-level Scheduler*). Isso implica na direta interferência do *scheduler* do *Hypervisor*, nos *schedulers* individuais de cada VM.

Figura 10 - Aninhamento de *Schedulers* em um sistema com *Hypervisor*.




Fonte: DE BOCK, 2020.

Para VMs executando sistemas GPOS a questão de aninhamento não é um problema, pois qualquer interferência, e consequentemente um atraso, não se transcreve como uma falha crítica. Entretanto, para o caso de um sistema RTOS, os prazos de execução devem ser respeitados com a maior precisão possível. Portanto, é indispensável que o nível de interferência entre os *schedulers* seja reduzido ao máximo.

Para contemplar este objetivo, é possível utilizar um recurso implementado pelo Xen chamado “*CPU Pinning*”. Nesta abordagem um núcleo físico de processamento (pCPU) é dedicado a um núcleo virtual (vCPU). Desta forma, o *scheduler* do hypervisor não vai interromper a execução de uma “vCPU” para alocar outra “vCPU”, em uma mesma “pCPU”. É importante ressaltar que mesmo removendo uma das “pCPUs” do ciclo de rotatividade empregado pelo *scheduler* do *Hypervisor*, no caso de um evento inesperado, como em uma *exception* que necessita de tratamento em EL2, o *scheduler* do *hypervisor* vai atuar e vai gerar atrasos.

A ideia é que o sistema RTOS execute operações críticas, isto é, que sejam resilientes e em menor número. Portanto, o número de ocorrências de *exceptions* que necessitem de tratamento em EL2 devem ser consideravelmente reduzidas se comparadas à um GPOS.

## 4 EMULAÇÃO

A implementação de um sistema para a validação das propostas foi feita através da emulação de um processador ARM Cortex-A57 (ARMv8-A 64 bits) utilizando o software QEMU. Todos os componentes foram compilados em um sistema de arquitetura x64, entretanto, como o objetivo é

executá-los em um sistema de arquitetura ARM, foi utilizado um processo de compilação cruzada através do utilitário de compilação *aarch64-linux-gnu-gcc*.

### 4.1 Bootloader

Após a inicialização da CPU emulada (processo realizado pelo próprio QEMU), o primeiro software executado é o *bootloader*[17]. O software *U-Boot* foi o escolhido como bootloader para integrar o ambiente de testes. Trata-se de um bootloader bastante difundido para aplicações em ambientes embarcados. O *U-Boot* é executado pelo QEMU e é responsável por carregar os binários pertinentes ao próximo estágio de inicialização para a memória, para enfim transferir a execução para o Xen.

### 4.2 Devicetree

Conforme requisitado pela especificação do Xen, é necessário provisionar uma *devicetree* que descreve o hardware existente (que neste caso está sendo emulado), bem como os periféricos que vão ser utilizados. A *devicetree* é definida caracterizando o mapa do espaço de memória e os barramentos controlados pela CPU. Posteriormente este arquivo é compilado para que possa ser lido pelo *U-Boot* como um binário no momento da inicialização. Vale ressaltar que é neste arquivo que é definido a instância dos diversos domínios (Dom0 e DomU), bem como a especificação para o *CPU Pinning*, caso um dos domínios seja destinado a execução um RTOS.

### 4.3 Instancias Linux

O kernel executado pelo Dom0 deve ser capaz de interagir com os componentes do hypervisor e executar as operações de VMI. Para criar uma imagem com os componentes de software necessários, foi utilizado a ferramenta Yocto Project. Esta ferramenta de código aberto, auxilia a criação de sistemas linux customizáveis para diversas arquiteturas. A grande vantagem é a simplificação do processo de integração de diversos componentes e aplicação de configurações necessárias, processos estes que passam a ser quase completamente automatizados pela ferramenta.

O processo de customização dos componentes que vão estar presentes dentro de uma imagem é feito através da definição de camadas, chamadas de “BBLAYERS”. A principal camada adicionada foi a “meta-virtualization”, ela possui vários componentes e utilitários para virtualização, entre elas: “xen-tools”, “libVMI” e as *shared libraries* do Xen.

A instância do DomU não precisa ter as mesmas capacidades do Dom0, portanto, pode ser mais leve e otimizada para diferentes tarefas. O Alpine Linux é uma distribuição Linux leve e orientada para a segurança, baseada no *busybox*[18]. Ela foi utilizada como base para testes do DomU. Na Figura 11 é apresentado o mapa da memória adotado para o carregamento de todos os arquivos e binários que são utilizados no sistema.

[17] Software responsável pela inicialização do sistema operacional.
[18] O Busybox é um software que combina versões de muitos utilitários UNIX comuns em um único pequeno executável.

Figura 11 - Mapa de organização da memória utilizado para a emulação do sistema.

| Componente | Endereço Inicial | Endereço Final |
|---|---|---|
| DOM0-ROOTFS | 0x417C29EA | 0x44000000 |
| DEVICETREE | 0x44000000 | 0x44001EFA |
| DOM0-KERNEL (Yocto) | 0x47000000 | 0x47E9FA00 |
| XEN-KERNEL | 0x49000000 | 0x490D0550 |
| DOMU-KERNEL (Alpine) | 0x53000000 | 0x5380DC38 |
| DOMU-ROOTFS | 0x58000000 | 0x5811AF01 |

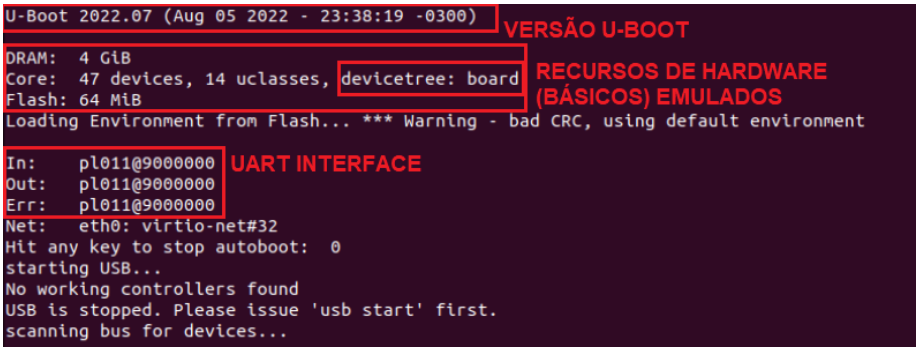
Fonte: Elaboração Própria.

Para ambas as instâncias são carregados os respectivos arquivos “rootfs”. Trata-se do sistema base de arquivos sobre o qual todos os demais arquivos e binários serão posteriormente montados. Vale ressaltar que a *devicetree* já está no formato de um binário, pois mesmo sendo definida em texto simples, para ser carregada e utilizada pelo sistema, ela deve ser compilada utilizando um compilador próprio chamado de *device tree compiler* (DTC).

### 4.4 Execução

Utilizando o próprio terminal da distribuição Linux, provisionada como ambiente de testes e desenvolvimento, é possível executar o QEMU especificando os parâmetros necessários para carregar em memória todos os componentes do sistema e dar início ao processo de boot. Na Figura 12 é apresentada uma captura de tela referente às saídas de texto apresentadas pelo bootloader U-Boot.

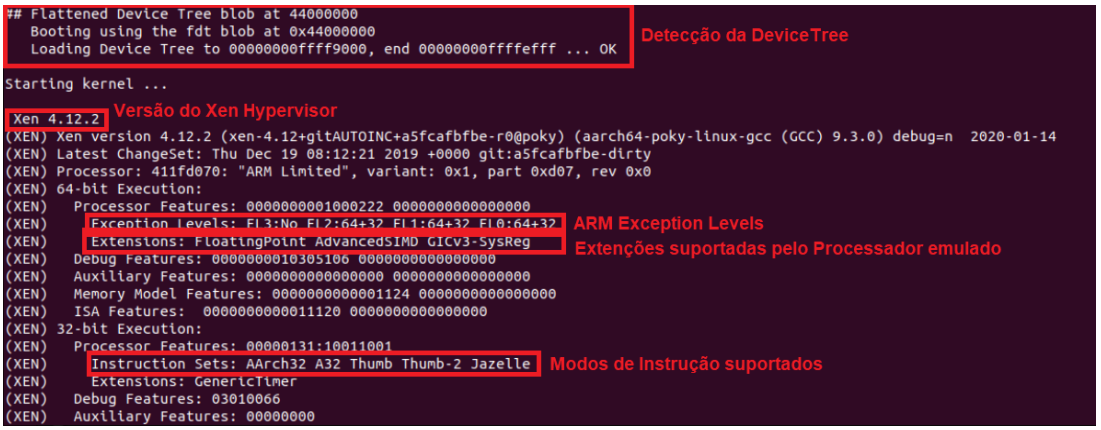
Figura 12 - Saída de dados do bootloader (U-Boot v2022.07) .




Fonte: Elaboração Própria.

É possível identificar que o bootloader já reconhece a existência de uma *devicetree*, bem como algumas características dos recursos do hardware emulado. O dispositivo “*pl011”* referenciado respectivamente como *input*, *output*, e *error* é a interface UART. Neste caso, a interface UART está sendo redirecionada diretamente para o terminal. Quando o bootloader termina seu próprio estágio de inicialização, o mesmo transfere o fluxo de execução para o Xen, que realizará sua própria inicialização, bem como a detecção e configuração de periféricos. O início desse processo pode ser observado na Figura 13.

Figura 13 - Saída de dados durante a inicialização do Xen.

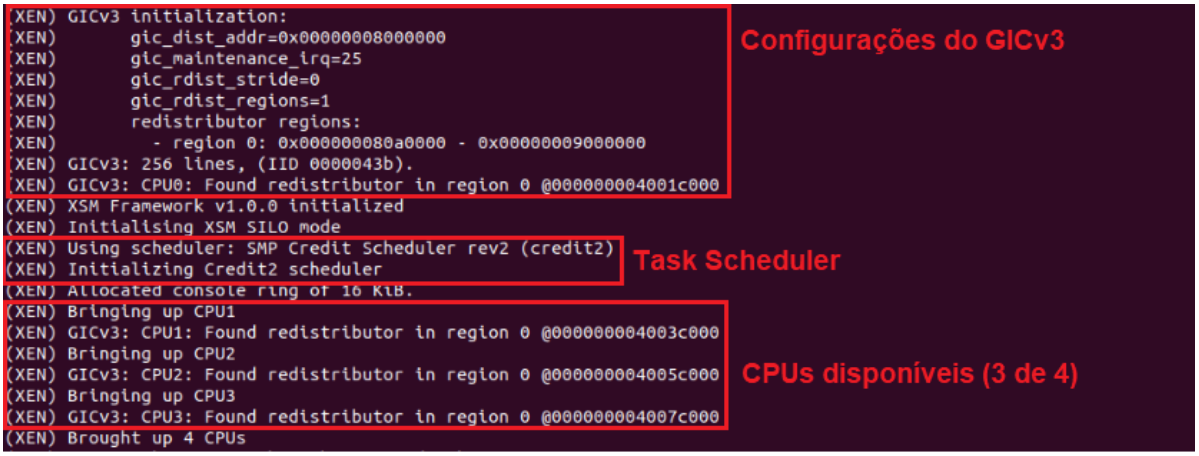



Fonte: Elaboração Própria.

É possível observar que o fluxo de execução já foi transferido para o Xen, que imediatamente inicia as rotinas para detecção do hardware disponível, enumerando os periféricos e suas respectivas regiões de memória e recursos. Os dados apresentados trazem informações importantes sobre o hardware que está sendo emulado. É pertinente destacar a versão do Hypervisor utilizada, bem como a detecção do suporte do processador aos quatro *Exception Levels* e do *GICv3*, que são indispensáveis para a virtualização.

Em um segundo momento, o Xen vai enumerar os núcleos de CPU disponíveis e iniciar o *scheduler* para organizar os tempos de execução de cada processo nos diferentes núcleos. O gerenciador de interrupções é configurado. Estes processos são mostrados pelo terminal, conforme mostrado pela Figura 14.

Figura 14 - Saída de dados durante a inicialização do Xen.

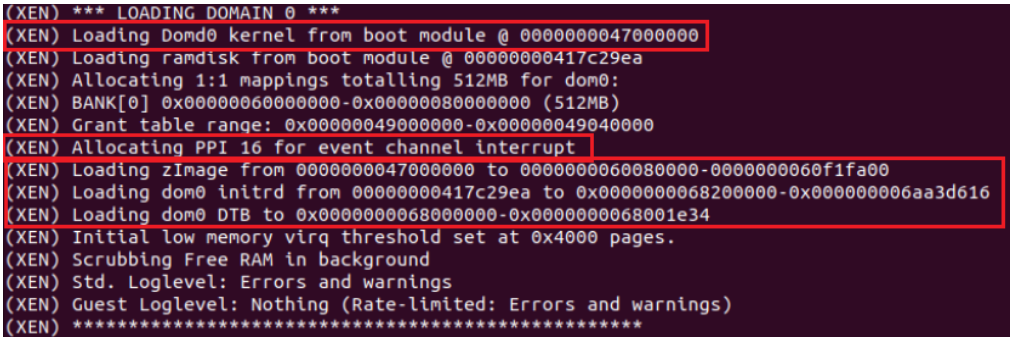



Fonte: Elaboração Própria.

É possível observar que o sistema possui quatro núcleos de CPU disponíveis, o que corrobora com a exigência para execução de um RTOS, já que neste caso, o sistema precisa de mais de um núcleo, para que ao menos um núcleo seja dedicado para a execução de um RTOS utilizando a técnica de "*CPU Pinning*" como já explicado. Ao todo existem quatro núcleos sendo emulados, cenário próximo ao se utilizar um processador real.

Após terminar a inicialização do *hardware*, o Xen inicia o processo de *boot* do *Dom0*. Durante esse processo, o binário da imagem do Kernel e dos sistema de arquivos do Dom0 são carregados para a memória. Também é alocada memória para o *event channel* para roteamento das interrupções e *exceptions*. Na Figura 15 é possível observar a saída de dados no terminal que corresponde a este processo.

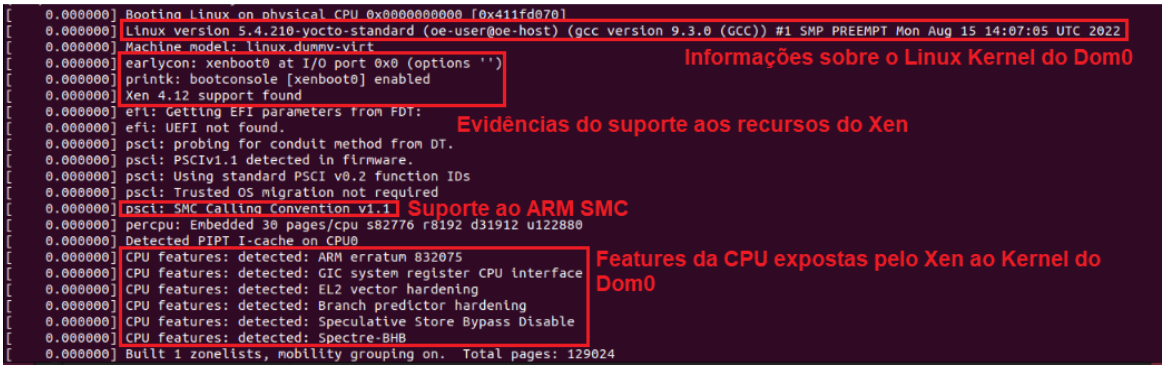
Figura 15 - Saída de dados durante a inicialização do Dom0.




Fonte: Elaboração Própria.

O processo de boot do Dom0 inicia com alocações especiais, para inicialização dos serviços internos do Xen que conferem ao Dom0 privilégios especiais. Em seguida, como mostrado na Figura 16, ocorre o boot do Linux presente no Dom0.

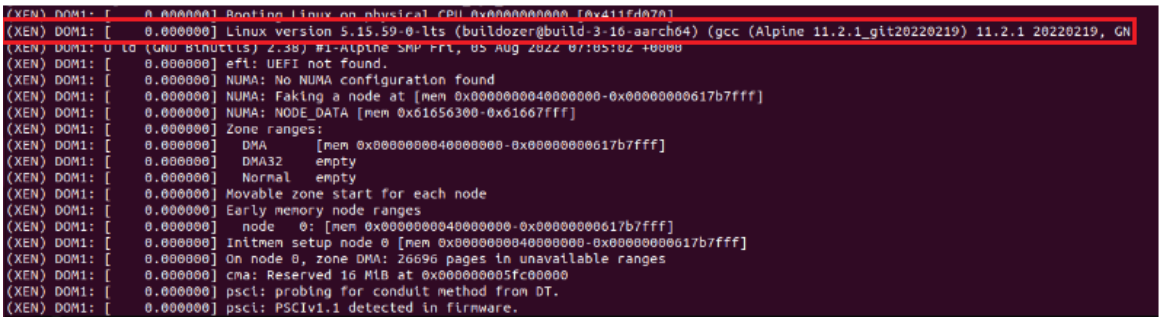
Figura 16 - Saída de dados durante a inicialização do Linux em execução no Dom0.




Fonte: Elaboração Própria.

É possível observar que o Linux provisionado pelo Yocto Project é inicializado corretamente como Dom0, o que é evidenciado pela detecção e configuração de recursos exclusivos do Kernel em execução no Dom0. Além disso, é importante observar o suporte à convenção de chamadas utilizando a instrução SMC, que é essencial para a implementação do sistema de *hooking* das *syscalls*, como já apresentado. Após a inicialização bem sucedida do Dom0, o Xen automaticamente prossegue para a inicialização do DomU (Dom1), como apresentado na Figura 17.

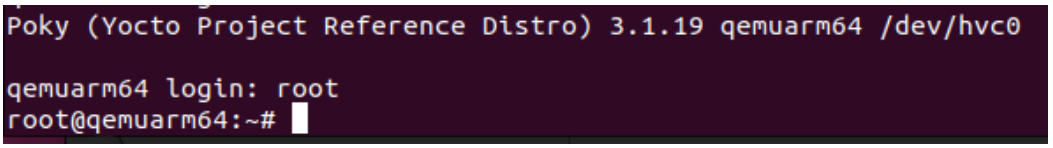
Figura 17 - Saída de dados durante a inicialização do Linux em execução no DomU (Dom1).




Fonte: Elaboração Própria.

Além das informações padrões do processo de boot do Linux, vale ressaltar que o Kernel utilizado, como esperado, é o Linux Alpine, sendo de fato, diferente da utilizada pelo Dom0. Além disso, não são apresentados processos que evidenciem a presença do Xen no sistema, isto é, do ponto de vista do Linux Alpine, este está sendo executado em um ambiente "nativo" (não emulado). Após a inicialização bem sucedida do DomU, o sistema apresenta a tela padrão de login do sistema para que o usuário possa ingressar como administrador do Hypervisor, obtendo controle sobre o Dom0, como apresentado na Figura 18.

Figura 18 - Tela de login do usuário administrador do Hypervisor (Dom0).




Fonte: Elaboração Própria.

Para alternar entre domínios, e passar a enviar comandos em DomU (Dom1), o usuário apenas precisa pressionar as teclas "CTRL + A" três vezes seguidas. O mesmo vale para retornar ao Dom0. Vale ressaltar que, como esperado, independente da tela apresentada ao usuário os dos Sistemas Operacionais continuam em execução paralelamente. Isto é, caso o usuário esteja executando operações através da linha de comando em Dom0, o sistema em DomU continua a execução.

## 5 RESULTADOS

Aplicações de VMI podem possuir graus de complexidade variáveis, mas a capacidade do processador em lidar com a virtualização determina quais são os horizontes de ação de um desenvolvedor. Este trabalho apresenta o potencial notável dos processadores ARM neste segmento e entrega a implementação de um ambiente completo, com muitos utilitários, para o desenvolvimento de aplicações de VMI. Além disso, também apresenta meios de realizar as operações de introspecção para monitoramento de ameaças que se encontram no estado da arte, compreendendo sistemas operacionais de aplicações gerais (GPOS) e aplicações críticas (RTOS).

É razoável pressupor que o desenvolvedor e usuário deste ambiente tenha conhecimentos referentes a arquitetura ARM, entretanto, o tempo gasto para configurar uma plataforma para o ciclo de testes, é grande. No momento da publicação deste artigo, não existem ambientes de emulação de processadores ARM que entreguem as funcionalidade de VMI diretamente (como a publicada junto deste artigo), isto é, que não necessitem de grandes esforços do usuário e investimento de tempo para configurar o arranjo de componentes necessários para iniciar o fluxo de desenvolvimento propriamente dito.

A disponibilização de uma plataforma como essa remove uma grande barreira de entrada para o desenvolvimento de softwares voltados para VMI, uma vez que qualquer desenvolvedor em potencial pode iniciar o ciclo de qualquer projeto, que tange o estado da arte, sem ter que se preocupar com detalhes de configuração e integração de componentes do ambiente. Podendo, portanto, trabalhar com níveis de abstração maiores, podendo se limitar e dar o foco apropriado às bibliotecas ou frameworks substanciais, como a "LibVMI". É factível portanto pontuar que o projeto do ambiente de emulação implementado é pioneiro na esfera de publicações de código aberto e pode ser encontrado através do repositório "*Sargastico/K3t4m1n3"*, no GitHub.

## REFERÊNCIAS

ARM (org.). **Learn the architecture - AArch64 Virtualization:** Stage 2 translation. [S. l.], 2019a. Disponível em: https://developer.arm.com/documentation/102142/0100/Stage-2-translation. Acesso em: 7 maio 2022.

ARM (org.). **Learn the architecture - AArch64 Virtualization:** Virtualization in AArch64. [S. l.], 2019b. Disponível em: https://developer.arm.com/documentation/102142/0100/Virtualization-in-AArch64. Acesso em: 7 maio 2022.

ARM (org.). **AArch64 Exception and Interrupt Handling:** AArch64 exception vector table. [S. l.], 2019c. Disponível em: https://developer.arm.com/documentation/100933/0100/AArch64-exception-vector-table. Acesso em: 7 maio 2022.

ARM (org.). **Learn the architecture - AArch64 Virtualization:** Secure virtualization. [S. l.], 2019d. Disponível em: https://developer.arm.com/documentation/102142/0100/Secure-virtualization. Acesso em: 7 maio 2022.

DE BOCK, Yorick; MERCELIS, Siegfried; BROECKHOVE, Jan; et al. Real-time virtualization with Xvisor. **Internet of Things**, v. 11, p. 100238, 2020. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S2542660520300718>. Acesso em: 16 jun. 2022.

DEVICETREE (org.). **DeviceTree Specification Release**. 0.3-40-g7e1cc17. [s.l.]: DeviceTree, 2021. Disponível em: https://github.com/devicetree-org/devicetree-specification/releases/download/v0.4-rc1/devicetree-specification-v0.4-rc1.pdf. Acesso em: 7 maio 2022.

DIGIKEY ELECTRONICS (org.). **Getting Started with STM32:** Introduction to FreeRTOS. [S. l.]. Disponível em: https://www.digikey.com.br/en/maker/projects/getting-started-with-stm32-introduction-to-freertos/ad275395687e4d85935351e16ec575b1. Acesso em: 7 maio 2022.

LENGYEL, Tamas K. **Stealthy monitoring with Xen altp2m**, 2016 Disponível em: <https://xenproject.org/2016/04/13/stealthy-monitoring-with-xen-altp2m/>. Acesso em: 7 maio 2022.

SIEWERT, Sam; PRATT, John. **Real-time embedded components and systems using Linux and RTOS**. Dulles, Virginia: Mercury Learning and Information, 2016.

SLOSS, Andrew N; SYMES, Dominic; WRIGHT, Chris. **ARM system developer's guide designing and optimizing system software.** Estados Unidos: Elsevier/ Morgan Kaufman, 2004.

TANENBAUM, Andrew S. **Modern operating systems**. Quarta edição. Boston: Pearson, 2015.

XEN PROJECT (org.). **Xen ARM with Virtualization Extensions whitepaper**. [S. l.], 2018a Disponível em: https://wiki.xenproject.org/wiki/Xen_ARM_with_Virtualization_Extensions_whitepaper. Acesso em: 7 maio 2022.

XEN PROJECT (org.). **Xen PCI Passthrough**. [S. l.], 2019a. Disponível em: https://wiki.xenproject.org/wiki/Xen_PCI_Passthrough. Acesso em: 7 maio 2022.

XEN PROJECT (org.). **Xen Project Schedulers**. [S. l.], 2019b. Disponível em: https://wiki.xenproject.org/wiki/Xen_Project_Schedulers. Acesso em: 7 maio 2022.

XEN PROJECT (org.). **Xen Project Software Overview**. [S. l.], 2018b. Disponível em: https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview. Acesso em: 7 maio 2022.