Deteção de máquinas virtuais

Pedro Miguel Nicolau Escaleira
escaleira@ua.pt

03/05/2021

Conteúdo

1 Introdução

Este trabalho foi feito no enquadramento dos temas lecionados na disciplina de Ambientes de Execução Seguros, do Mestrado de Cibersegurança da Universidade de Aveiro. Nele, foi-nos proposto criarmos um programa que possuísse "consciência"de se está a ser ou não executado dentro duma máquina virtual.

No ponto de vista de segurança informática, existem múltiplos casos de uso onde este tipo de programas podem ser usados, desde *malware* que altera o comportamento a partir do momento que detete que está a ser executado numa máquina virtual, dificultando a sua deteção e análise de comportamento quando executado num ambiente seguro como uma *sandbox* [1], até *rootkits* que virtualizam todo o sistema operativo sob ataque, permitindo ganhar controlo do mesmo. Estes *rootkits*, ao virtualizarem todo o sistema sob ataque, podem ser difíceis de detetar, sendo importante programas como os desenhados durante este trabalho para alcançar esse objetivo.

O primeiro *rootkit* desta classe que criado, foi o projeto *SubVirt* da *Microsoft* [2], que fazia uso de "virtualizadores comerciais", como o *VMware* para fazer a virtualização do sistema operativo. Contudo, pelo facto de usar este tipo de virtualização, a sua instalação num sistema era complexa e, sendo que tinha de haver uma emulação de todo o *I/O*, a deteção era mais ou menos facilitada, por exemplo através da deteção de modificações das *drivers* do SO. Mais tarde, surgiram *rootkits* desta familia mais complexos que o *SubVirt*, como por exemplo o *Blue Pill*, desenhado pela investigadora *Joanna Rutkowska*, que fazia virtualização ao nível do *hardware*, aumentando a complexidade da sua deteção, e que era muito mais "leve"do que o primeiro. Neste caso, um dos métodos que poderia ser usado para detetar a existência de virtualização é a análise de quanto tempo demoram certas instruções a serem executadas[1], tendo sido este o estudado e desenvolvido neste trabalho. [3, 4]

[1] Acima foi explicado que uma das formas de facilmente detetar a presença do *SubVirt* seria analisar alterações das *drivers*. Contudo, o *Blue Pill* não fazia qualquer alteração das mesmas, uma vez que virtualizava o sistema operativo ao nível do *hardware*, exigindo um mecanismo de deteção mais complexo e diferente deste primeiro.

2 Medições

O primeiro requerimento que tivemos de ter em consideração, antes de fazer qualquer medição, foi como é que iríamos medir a passagem de tempo duma dada instrução. Se, por um lado, fizéssemos uma medição com base no relógio do sistema, significaria que o valor das medições seriam demasiado dependentes da velocidade do CPU e do sistema operativo do computador onde seriam feitas. Contudo, se por outro lado fizéssemos uma medição com base em **ciclos de relógio** decorridos durante a execução da instrução sob estudo obteríamos, teoricamente, resultados dependentes da arquitetura do CPU. Desta forma, foi esta ultima estratégia a adotada no nosso trabalho, uma vez que é a mais consistente.

Apesar de ter sido dito, no parágrafo anterior, que com uma medição com base nos **ciclos de relógio** obteríamos medições dependentes apenas da arquitetura do CPU, na prática isto não é inteiramente verdade. Uma das razões é porque não há maneira de obtermos empiricamente o número exato de ciclos de relógio duma dada instrução. O mais próximo que temos deste conceito é usarmos a instrução ***rdtsc*** [5], ou a ***rdtscp*** [6], usadas para obter o valor do *Time Stamp Counter* (TSC) [7], que é o registo onde se encontra armazenado o número de ciclos desde o ultimo *reset*, caso o CPU estivesse a operar constantemente à velocidade máxima. Ou seja, obtendo este valor no inicio e no final duma dada instrução ter sido executada, somos contemplados com uma aproximação de quantos ciclos de relógio essa instrução levou até ser completada (caso o CPU esteja a operar à frequência máxima).

Outro dos principais motivos pelo qual não é possível obter uma medição precisa do número de ciclos de relógio duma dada instrução, é devido à existência dum *process scheduler* nos sistemas operativos modernos que, resumidamente, determina que "porção"dum dado processo será executada em cada momento no processador. Isto permite a "divisão"dum processo em várias partes, sendo que cada uma não é necessáriamente executada a seguir à anterior, uma vez que o *scheduler* poderá indicar que outra "parte"de outro processo seja executada entretanto, colocando o nosso processo em espera. Desta forma, sendo que são executadas múltiplas instruções para fazermos a medição dos ciclos de relógio duma determinada instrução (como vimos acima, pelo menos é necessário executar duas vezes uma das duas instruções de leitura do TSC, para além da própria instrução que está a ser estudada), não há maneira de garantir que elas sejam realmente realizadas uma a seguir à outra [8, 9]. De forma a diminuir o erro que é introduzido devido à existência deste mecanismo, as instruções sob estudo foram executadas múltiplas vezes, de forma a obter o máximo de medições possíveis, na expectativa de que quantas mais vezes a instrução for executada, mais provável será encontrar o mínimo de número de ciclos de relógio que ela necessita para ser executada.

Outro ponto importante a notar sobre este estudo foi que tentamos usar instruções que possívelmente causariam ***VM Exits***. Este evento ocorre quando a máquina virtual necessita de executar certas instruções específicas, que não podem ser executadas diretamente no CPU, sendo que há a "passagem do controlo"para o *Virtual Machine Monitor (VMM)*, de forma a que a instrução seja virtualizada pelo *Hypervisor* [10]. Devido ao tempo acrescido do ***VM Exit*** e do consequente ***VM Entry*** conseguimos, em teoria, detetar se o nosso programa está a ser executado numa máquina virtual, por comparação do tempo que uma dada instrução demoraria num ambiente não virtualizado (onde estes eventos não ocorrem e a instrução é executada diretamente no CPU).

Contudo, sendo que uma instrução é executada muito rápidamente por um processador, mesmo quando num ambiente virtualizado, fizemos as nossas medições perante o tempo que demorava a ser executada essa instrução num ciclo, por exemplo de 100 iterações. Por outro lado, após aviso dos docentes, foi-nos chamado a atenção de que esta medição não bastaria, uma vez que, mesmo para CPUs com a mesma arquitetura, haveriam variações no tempo que uma mesma instrução demoraria a ser executada. Desta forma, tivemos de acrescentar ás nossas medições algo que nos permitisse fazer a deteção em máquinas diferentes com processadores da mesma arquitetura. O método levado a cabo foi utilizar uma segunda instrução para controlo, no nosso caso a instrução *XOR*, que não provoca qualquer *VM Exit* nas máquinas virtuais[2]. Desta forma, tendo em conta que o número de ciclos de relógio que as instruções de duas máquinas com CPUs distintos, mas da mesma arquitetura, variam da mesma forma de um CPU para o outro, a razão entre os ciclos de relógio das duas instruções é similar entre as duas máquinas. Ou seja, se tivermos uma instrução *A*, que demore *x* vezes mais ciclos de relógio numa dada máquina do que noutra, com CPUs diferentes, mas

[2] Isto permite que, para uma mesma máquina, o valor mínimo do tempo que um ciclo com as mesmas iterações desta instrução seja sensivelmente o mesmo.

da mesma arquitetura, outra instrução *B* demorará também sensivelmente *x* vezes mais ciclos de relógio na primeira máquina que na segunda. Sendo assim, a razão entre os ciclos de relógio das instruções *A* e *B* será similar nas duas máquinas.

Ainda sobre a implementação dos programas usados para fazer as medições, foi acima referido que foi feito um ciclo de instruções *XOR*, cujo número de iterações foi igual ao número de iterações do ciclo de instruções da instrução sob estudo. Contudo, caso indiquemos ao nosso programa para executar o mesmo *XOR* sob os mesmos registos num ciclo, o *pipeline* do CPU irá "ignorar"parte destas instruções, porque os últimos *XORs* anulam o que os anteriores fariam sob esses registos, uma vez que são todos iguais. Isto teria como consequência resultados incongruentes entre si, uma vez que não estaria no nosso controlo o número de instruções *XOR* que o CPU iria executar. Desta forma, usámos um conjunto de 4 instruções *XOR* "hostis à *pipeline*"[11], providenciadas pelos docentes, encadeadas de tal forma que cada uma depende do resultado da instrução *XOR* anterior, impedindo a *pipeline* de "simplificar o trabalho". Assim sendo, este conjunto de 4 instruções *XOR* foi colocado dentro do referido ciclo, permitindo fazer a razão entre o tempo que este ciclo demora e o correspondente ciclo da instrução em estudo.

Em termos do *setup* usado para fazer o nosso estudo, usamos duas máquinas com processadores de duas marcas distintas, *AMD* e *Intel*, ambos *X86*, de forma a podermos validar a nossa hipótese de que a razão entre os ciclos de relógio duma dada instrução sob estudo e a instrução *XOR* seria similar entre processadores distintos da mesma arquitetura. Em cada uma das máquinas, usamos o *Manjaro Arch Linux* como sistema operativo para fazer a execução dos nossos programas de medição e *setup* das máquinas virtuais. Em cada um deles, as máquinas virtuais usadas para fazer o nosso estudo foram a *VirtualBox*, *VMware* e *QEMU*, todas com o *Ubuntu 20.04 LTS* instalado. Na máquina com o processador da *Intel*, sendo que possuíamos o *Windows* em *dual boot*, fizemos também medições numa máquina virtual *Hyper-V*, também com o sistema operativo *Ubuntu 20.04 LTS*. Desta forma, pudemos obter uma gama variada de resultados para diferentes ambientes de execução, possibilitando um estudo mais abrangente.

2.1 Medições em modo de utilizador

Como explicado anteriormente, fizemos as nossas medições usando o método de executar um ciclo com um conjunto de instruções *XOR*, seguidas dum outro ciclo, com o mesmo número de iterações do primeiro, mas que executa, neste caso, a instrução *CPUID* [12]. Esta instrução foi escolhida para ser usada em modo de utilizador, uma vez que, de acordo com o manual da *Intel*[3] [13], esta é uma instrução que pode ser executada em modo de utilizador i.e., no modo protegido *Ring 3* do processador, e que provoca **incondicionalmente** um ***VM Exit*** quando executada numa máquina virtual.

Executamos, desta forma, este conjunto de etapas num ciclo com tamanho de 1000000, nas duas máquinas e respetivos ambientes virtuais e não virtuais referidos. Depois de obtermos os mínimos dos tempos obtidos para o conjunto de instruções *CPUID* e *XOR*, fizemos a razão entre estes dois valores para cada uma das diferentes execuções e obtivemos os resultados demonstrados no gráfico da figura 1.

Da análise deste gráfico é possível fazer algumas conclusões:

- A diferenciação entre um ambiente virtual e não virtual é notável, uma vez que a razão dos tempos no *host* é muito menor do que em qualquer uma das máquinas virtuais.
- Não existe uma grande diferença entre os valores obtidos nos virtualizadores *QEMU* e *VirtualBox*, em qualquer um dos processadores, pelo que o *fingerprint* de máquinas virtuais irá, potencialmente, dar muitas suposições trocadas nestes dois ambientes.
- Os valores obtidos no *VMware* e *Hyper-V*, no processador da *Intel*, são também muito similares, podendo originar o mesmo problema referido no ponto anterior, mas entre estes dois virtualizadores.

Outra dedução que é possível fazer é que a nossa hipótese de que os valores da razão, quando o programa executado num mesmo ambiente, variam muito pouco substancialmente entre os dois CPUs, uma vez que possuem a mesma arquitetura (X86).

[3] Na secção 25.1.2 do manual.

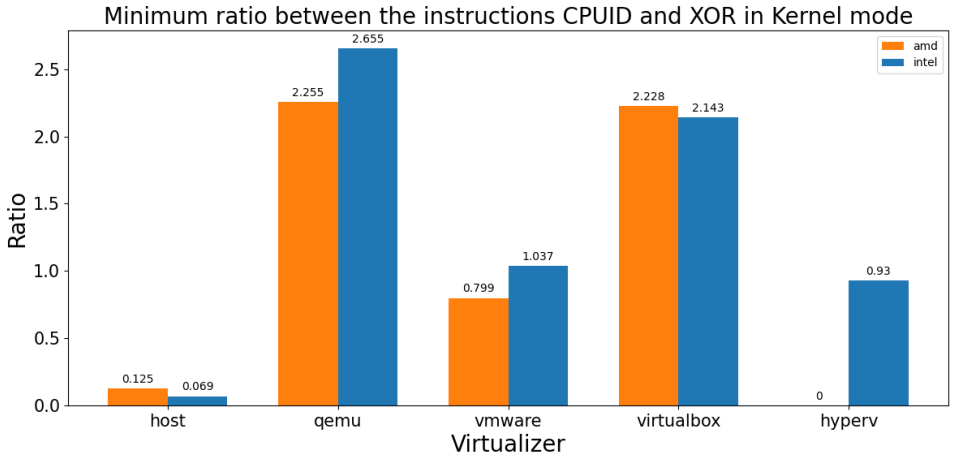

Figura 1: Razão entre o tempo de execução do conjunto de instruções *CPUID* e *XOR* em modo de utilizador.

2.2 Medições em modo kernel

Tal como nas medições em modo de utilizador, utilizamos o mesmo método de medição, mas desta feita, em vez de apenas fazermos medições com a instrução *CPUID*, usamos outras duas que provocavam ***VM Exits* condicionais**: o par ***SGDT*/*LGDT***, usadas para obter e guardar valores na *global descriptor table* [14, 15], e o par ***OUTb*/*INb***, no nosso caso usadas para obter o valor em segundos do *Real Time Clock (RTC)* [16, 17, 18]. Este conjunto de instruções só podem ser executadas no modo protegido *Ring 0* do processador, ou seja, em modo *kernel*. Como a própria expressão "condicionais"indica, estas instruções não garantem que haja um *VM Exit* sempre que sejam executadas, sendo que isso depende de certas configurações da própria máquina virtual. Poderíamos, claro, ter usado uma instrução de *Ring 0* que provocasse um ***VM Exit* incondicional**, como a instrução *INVD*, recomendada pelos professores da disciplina. Contudo, esta era demasiado perigosa se operada por alguém inexperiente nesta matéria, como era o caso, uma vez que invalida as *caches* internas do processador, pelo que decidimos não a usar [19].

Desta forma, criamos um módulo de *kernel*, de forma a podermos executar estas 3 instruções/par de instruções em *Ring 0*. Com este, coletamos dados do tempo de execução do conjunto de *XORs* e de cada conjunto de cada uma destas instruções, num ciclo de tamanho 10000 cada. No final, obtivemos as razões entre os mínimos dos tempos de cada instrução e do correspondente mínimo de tempo que o conjunto de *XORs* demorou a ser executado. Os resultados obtidos, para cada uma das instruções, podem ser encontrados nos gráficos das figuras 2, 3 e 4.

Mais uma vez, depois da análise destes dados obtidos, foi-nos possível fazer algumas conclusões:

- No caso da instrução *CPUID*, os resultados obtidos foram, como expectável, similares aos resultados obtidos quando fizemos os mesmos testes em modo de utilizador. Contudo, no caso do *VMware*, foram algo inesperados, uma vez que para além de muito diferentes dos obtidos em *Ring 3*, foram também muito similares aos obtidos para um ambiente não virtualizado. Ora, tendo em conta que esta instrução causa um *VM Exit* incondicional, o esperado seria que os resultados num ambiente virtual, como uma máquina virtual *VMware*, fossem consideravelmente superiores aos obtidos num ambiente não virtualizado. Ao relatar estes resultados ao professor da disciplina André Zúquete, este informou que o *VMware* faz **tradução binária** (do inglês ***binary translation***) do código do *kernel*, mas não do código executado em modo de utilizador. Desta forma, poderá substituir a instrução *CPUID* "por um conjunto de atribuições a registos", sem necessidade de qualquer *VM Exit*. Após alguma pesquisa, encontrámos referência a esta técnica num dos *white papers* do *VMware*, onde descrevem que o código do *kernel* que possua instruções "não virtualizáveis"é substituído por código com instruções que tenham o mesmo resultado e que possam ser executadas no *hardware* virtualizado [20]. Para rematar, usando esta técnica de tradução binária em modo *kernel*, o *VMware* "consegue"substituir a instrução *CPUID* por um

conjunto de outras instruções que sirvam o mesmo propósito no ambiente virtualizado, evitando um *VM Exit* e um consequente *VM Entry*, sendo que desta forma obtivemos valores similares aos obtidos num ambiente não virtualizado. Por comparação entre os gráficos das figuras 1 e 2, é também possível reparar que, excetuando o caso da *VMware*, os resultados obtidos são similares aos obtidos com esta instrução em modo de utilizador.

- No caso das instruções do par *SGDT*/*LGDT* pode-se concluir, após observação do gráfico da figura 3, que não será possível, pelo menos de forma precisa, fazer deteção de se o nosso programa está a ser executado num ambiente virtualizado ou não, nem o *fingerprinting* dos respetivos virtualizadores. Isto porque, apesar destas instruções provocarem um *VM Exit* condicional, nas nossas medições obtivemos resultados muito parecidos entre todos os ambientes, pelo que podemos concluir que em todos eles não houve *VM Exit*, tendo ocorrido qualquer outro mecanismo como o já discutido tradução binária, que permitiu que as razões entre os tempos medidos fossem muito similares ás obtidas num ambiente não virtualizado. É de notar, contudo, que no caso do *QEMU* na máquina com o processador da *Intel*, obtivemos uma razão entre os tempos mínimos muito superior a qualquer outro ambiente, mesmo com o mesmo virtualizador na máquina com o processador da *AMD*, sendo que a explicação mais apropriada é que talvez haja uma diferença na forma como a virtualização está a ser feita neste caso, mesmo entre o mesmo virtualizador mas para processadores diferentes.
- Quanto ás instruções do par *OUTb*/*INb*, cuja representação gráfica dos resultados pode ser observada no gráfico da figura 4, é de notar que há ambientes em que as razões dos tempos obtidas nos dois *CPUs* é muito díspar. No caso do ambiente não virtualizado (*host*), isto pode ser explicado porque, sendo que o conjunto de instruções usadas pede o valor do *RCT*, temos de ter em conta que há um tempo acrescido, dependente da velocidade do *hardware*, em que o CPU pede este valor e o recebe do *RCT*, pelo que deste gráfico se poderá concluir que o *hardware* que é usado para fazer o pedido deste valor e para enviar este para o *CPU* é mais lento na máquina com o processador da *Intel* do que na que possui o da *AMD*. Contudo, foi com alguma surpresa que observamos que, apesar de no *host* haver esta diferença, nas máquinas virtuais *QEMU* e *VMware* esta ocorreu ao contrário i.e., o valor obtido na máquina com o processador da *AMD* foi superior ao da máquina com o processador da *Intel*. Outra característica verificada foi que houve virtualizadores, principalmente na máquina com o processador da *Intel*, em que a razão obtida foi menor do que a obtida no ambiente não virtualizado, o que pode significar uma vez mais que não houve *VM Exit* (recordemo-nos que estas duas instruções não garantem um *VM Exit* em todas as circunstâncias), ou o *VMM* pode usar técnicas como a tradução binária para não executar as instruções dadas, mas sim outras que tenham o mesmo propósito, mas que possam ser executadas em *hardware* virtualizado. Tendo em conta todas estas diferenças entre CPUs e entre o *hardware* das máquinas, podemos concluir que este conjunto de instruções, pelo menos quando usadas para ler o *RTC*, poderão não ser a nossa melhor opção se o nosso intuito é detetar a presença de virtualização ou fazer *fingerprint* de máquinas virtuais.

Das observações feitas acima, podemos concluir que, em modo *kernel*, a instrução que oferece mais estabilidade de resultados para um mesmo ambiente e onde estes permitem uma melhor discriminação entre os vários ambientes é a *CPUID*, pelo que esta será a opção mais viável a usar na deteção de virtualização e *fingerprint* dos vários virtualizadores.

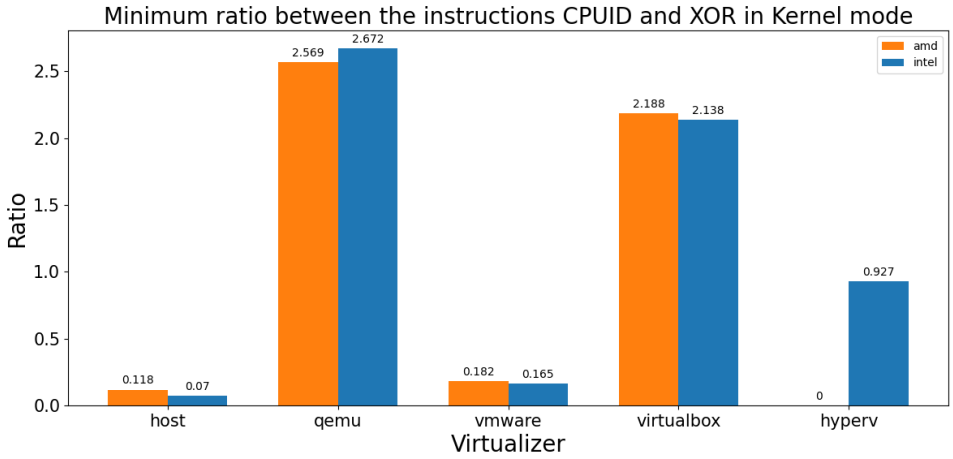

Figura 2: Razão entre o tempo de execução do conjunto de instruções *CPUID* e *XOR* em modo *kernel*.

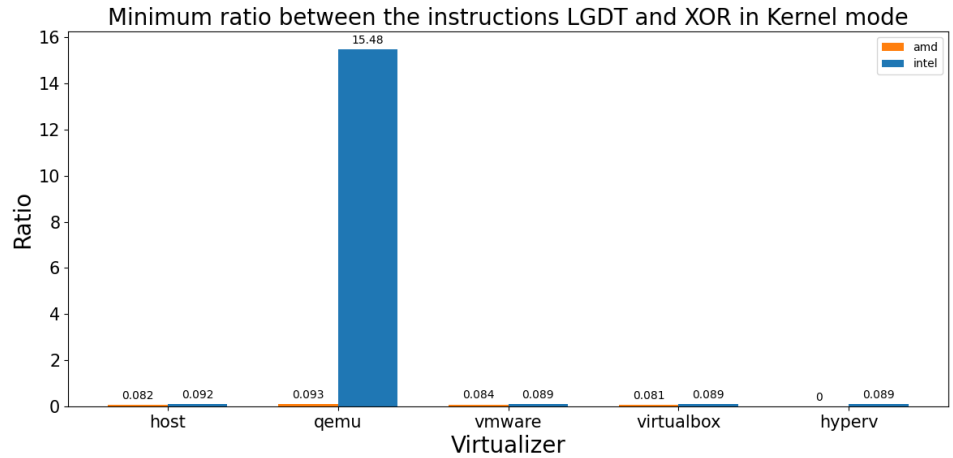

Figura 3: Razão entre o tempo de execução do conjunto de instruções do par *SGDT*/*LGDT* e *XOR* em modo *kernel*.

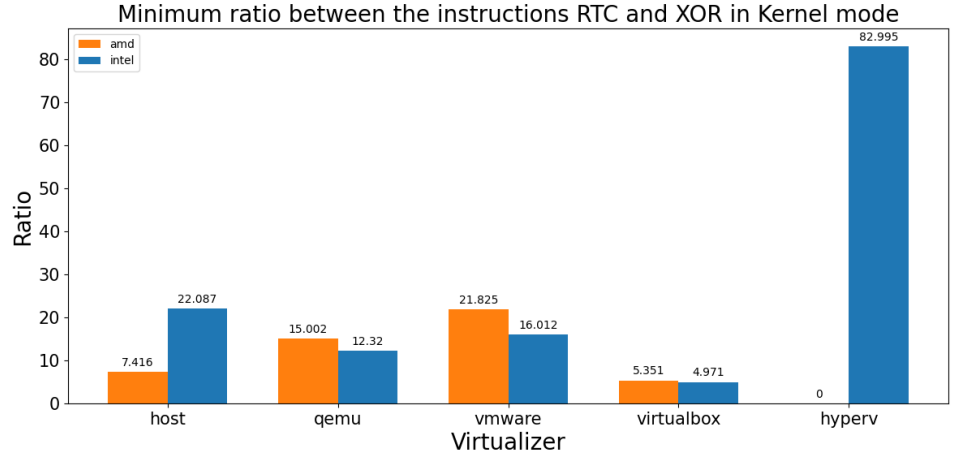

Figura 4: Razão entre o tempo de execução do conjunto de instruções do par *OUTb*/*INb* e *XOR* em modo *kernel*.

3 Deteção de virtualização

Depois das medições feitas e da correspondente análise dos dados obtidos, como apresentado na secção 2, prosseguimos para a deteção de virtualização e deteção de virtualizadores automáticas. Para isso, criamos um *script* de *Python* que indica qual o ambiente mais provável em que foi executado. Desta forma, podemos dizer que este código faz *fingerprinting* do ambiente em que é executado, pelo que os dois propósitos deste trabalho estão incluídos nele mesmo (ou seja, ao fazer *fingerprinting* do ambiente, se detetar que o ambiente é o *host*, significa que não detetou virtualização e caso indique um dos virtualizadores, detetou virtualização e fez o correspondente *fingerprint* do virtualizador).

O funcionamento deste código baseia-se, numa primeira etapa, na execução de um dos módulos apresentados na secção 2, de seguida, na obtenção dos resultados dessa mesma execução, filtragem dos mínimos temporais da execução da instrução *XOR* e de uma das instruções sob estudo e, por fim, faz uma comparação entre a razão destes dois mínimos e as razões obtidas durante a fase de medições. Desta comparação, que é feita através da diferença absoluta entre cada um dos valores previamente obtidos em cada ambiente e o valor obtido durante esta fase de deteção, o *script* seleciona a menor das diferenças e seleciona o ambiente que originou esse resultado como sendo o resultado da deteção.

O *script* é também versátil no que compete ao estudo que pretendemos fazer. Desta forma, podemos escolher a priori:

- A instrução a ser usada. Tendo em conta que fizemos medições para múltiplas instruções diferentes, é possível no detetor indicar qual a instrução a ser usada para fazer a deteção.
- Os dados de um dos CPUs usados nas medições previamente feitas. Sendo que fizemos medições usando duas máquinas diferentes com processadores de duas marcas diferentes, o *script* permite também selecionar quais os dados que pretendemos usar para fazer a comparação, se os obtidos com o processador da *AMD* ou com o da *Intel*.
- O modo em que pretendemos fazer a deteção. Podemos indicar se preferimos fazer a deteção em modo de utilizador ou em modo *kernel*.

É também possível indicar ao código, que pretendemos que ele dê um resultado mais verboso, sendo que neste caso irá não só indicar qual o ambiente em que "pensa"que foi executado, mas também os resultados das múltiplas comparações entre a medição feita no momento e as medições rótuladas. Na listagem 1 é apresentado o resultado da execução deste código com *output* verboso.

```
$ python hypervisor_detection.py --instruction cpuid --mode user --cpu amd --verbose 1



*********************************************************************************
Minimum obtained on this execution:              0.1105675

Distance to resuts from <host>:                  0.0139692           HIT
Distance to resuts from <qemu>:                  2.1448510
Distance to resuts from <vmware>:                0.6882005
Distance to resuts from <virtualbox>:            2.1177051
Distance to resuts from <hyperv>:                0.1105675
*********************************************************************************



Ambient detected using the data from the amd CPU with the instruction CPUID in user mode:       host
```

Listagem 1: Exemplo da execução do detetor.

Nas deteções feitas por nós, usamos apenas a instrução *CPUID*, quer em modo utilizador, quer em modo *kernel*, uma vez que, como explicado na secção 2, as outras instruções testadas apresentaram uma performance precária em termos de diferenciação de qual o ambiente em que foram executadas. Quanto aos

resultados propriamente ditos, em modo utilizador o detetor foi sempre capaz de detetar quando estava a ser executado num ambiente não virtualizado (ou seja, no *host*) e fez sempre corretamente a identificação do virtualizador no caso das execuções nos ambientes virtuais, falhando apenas a identificação quando usados os dados do processador da *Intel* nos ambientes *VMware* e *Hyper-V* e quando usados os dados do processador da *AMD* nos ambientes *QEMU* e *VirtualBox*, por vezes confundindo-os entre si. Já em modo *kernel*, as deteções feitas estiveram também dentro do esperado, sendo que por vezes houve a confusão entre se o código estaria a ser executado num ambiente virtualizado ou na máquina virtual da *VMware*, sendo que para os restantes ambientes conseguiu sempre fazer a identificação correta.

4 Referências

[1] Nishad Herath. Virtual machines and how malware authors know when they are being watched, Mar 2020. Disponível em: https://securityintelligence.com/virtual-machines-malware-authors-being-watched/ [Acedido em 12-05-2021].

[2] Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen Wang, and Jay Lorch. Subvirt: Implementing malware with virtual machines. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 314–327. Institute of Electrical and Electronics Engineers, Inc., May 2006. Disponível em: https://www.microsoft.com/en-us/research/publication/subvirt-implementing-malware-with-virtual-machines/ [Acedido em 12-05-2021].

[3] George Ou. Blue pill: The first effective hypervisor rootkit, Aug 2006. Disponível em: https://www.zdnet.com/article/blue-pill-the-first-effective-hypervisor-rootkit/ [Acedido em 12-05-2021].

[4] George Ou. Detecting the blue pill hypervisor rootkit is possible but not trivial, Aug 2006. Disponível em: https://www.zdnet.com/article/detecting-the-blue-pill-hypervisor-rootkit-is-possible-but-not-trivial/ [Acedido em 12-05-2021].

[5] Félix Cloutier. Rdtsc — read time-stamp counter. Disponível em: https://www.felixcloutier.com/x86/rdtsc [Acedido em 28-04-2021].

[6] Félix Cloutier. Rdtscp — read time-stamp counter and processor id. Disponível em: https://www.felixcloutier.com/x86/rdtscp [Acedido em 28-04-2021].

[7] Kazutomo Yoshii. Time-stamp counter. Disponível em: https://www.mcs.anl.gov/~kazutomo/rdtsc.html [Acedido em 28-04-2021].

[8] Operating system - process scheduling. Disponível em: https://www.tutorialspoint.com/operating_system/os_process_scheduling.htm [Acedido em 28-04-2021].

[9] Cfs scheduler. Disponível em: https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html [Acedido em 28-04-2021].

[10] Yasser Shalabi. The cost of virtualization exits. Disponível em: http://yshalabi.github.io/VMExits/ [Acedido em 02-05-2021].

[11] Detecting cpu speed - knowing how many cycles your loop takes. Disponível em: https://wiki.osdev.org/Detecting_CPU_Speed#Knowing_how_many_cycles_your_loop_takes [Acedido em 03-05-2021].

[12] Félix Cloutier. Cpuid — cpu identification. Disponível em: https://www.felixcloutier.com/x86/cpuid [Acedido em 28-04-2021].

[13] *Intel® 64 and IA-32 Architectures - Software Developer's Manual*, volume 3. Intel. Disponível em: https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-3a-3b-3c-and-3d-system-programming-guide.html [Acedido em 02-05-2021].

[14] Félix Cloutier. Sgdt — store global descriptor table register. Disponível em: https://www.felixcloutier.com/x86/sgdt [Acedido em 02-05-2021].

[15] Félix Cloutier. Lgdt/lidt — load global/interrupt descriptor table register. Disponível em: https://www.felixcloutier.com/x86/lgdt:lidt [Acedido em 02-05-2021].

[16] Rtc. Disponível em: https://wiki.osdev.org/RTC [Acedido em 02-05-2021].

[17] Ciro Santilli. x86-bare-metal-examples/in_rtc.s. Disponível em: https://github.com/cirosantilli/x86-bare-metal-examples/blob/9a24f92f36a45abb3f8c37aafc0c3ee9b15563ab/in_rtc.S [Acedido em 02-05-2021].

[18] Inline assembly/examples - i/o access. Disponível em: https://wiki.osdev.org/Inline_Assembly/Examples#I.2FO_access [Acedido em 02-05-2021].

[19] Félix Cloutier. Disponível em: https://www.felixcloutier.com/x86/invd [Acedido em 02-05-2021].

[20] *Understanding Full Virtualization, Paravirtualization, and Hardware Assist*, page 4–4. VMware. Disponível em: https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/VMware_paravirtualization.pdf [Acedido em 03-05-2021].