universidade de aveiro deti departamento de electrónica, telecomunicações e informática

Detection of Hypervisors

Cybersecurity MCs: Secure Execution Environments

David Gomes Morais, 93147

Junho, 2022
DETI, Universidade de Aveiro

Introdução

O presente relatório tem como objetivo explicar e documentar as decisões tomadas na implementação e uso de timing strategies por forma a detetar a presença de virtualização *stealthy* num ambiente através da análise de fenómenos inerentes ao processo de virtualização em si, que não podem ser facilmente escondidos do hosted system, desenvolvido no âmbito do primeiro projeto da cadeira de Ambientes de Execução Seguros.

Este processo foi baseado na noção de que um virtualized guest opera mais lentamente do que um native host, e uma estratégia inicial foi desenvolvida segundo tal e empregando instruções que causam uma trap no visualizador, nos casos em que um guest virtualizado passa o controlo ao host para executar uma dada operação de maneira a que o hypervisor tenha a possibilidade de emular data structures e comportamentos de um host nativo. São também apelidadas de VM Exits e podem ser explícitas, onde o hypervisor é chamado diretamente, mas as traps em que as estratégias desenvolvidas têm mais interesse é nos resultantes VM Exits implícitas, que ocorrem em certas situações, como por exemplo execução de certas instruções assembly. O desenvolvimento foi dividido em dois níveis de proteção - um a partir do ring 3 (user space) e uma a partir do ring 0 (através de um Linux Kernel Module) que faz uso de instruções privilegiadas.

Para tal, foi começado por se desenvolver uma estratégia inicial, fazendo uso de um programa em C, no ring 3, definindo estratégias para medição de tempos de execução precisos, assim como threshold para a detecção e fingerprinting de virtualização. Esta foi depois portada para poder ser usada no ring 0, através de um módulo do kernel como foi mencionado, fazendo uso de instruções privilegiadas. Além disso, no ring 3 foi modificada a implementação por forma a perceber o comportamento do hypervisor quando certas system calls são utilizadas em vez de instruções assembly. Os resultados foram obtidos e analisados e são discutidos neste relatório, como foco na diferença entre valores escolhidos para medição de elapsed time assim como nos tempos de execução em rings distintos.

Definição de uma estratégia

A estratégia utilizado utiliza um threshold para o qual, se o tempo de execução de uma VM Exit provocada por uma CPU instruction estiver acima do mesmo, então é considerado que o sistema é virtualizado, visto que em sistemas nativos esta VM Exit não ocorre fazendo com que o tempo de execução seja relativamente mais rápido. Existem várias instruções, privilegiadas e não privilegiadas, que podem causar VM Exists, mas o trapping de algumas pode ser disabled durante o setup da VM.

Como foi mencionado anteriormente, começou-se pelo user space, utilizando instruções assembly, primeiro trabalhando com os potenciais problemas e desafios de executar medições de tempos de execução precisas para instruções tão rápidas e depois focando-se nas instruções usadas e numa forma homogénea de detecção em vários tipo de hardware diferente.

Medição precisa de tempos de execução

O principal desafio nas medições de tempo foi o cuidado com que tais tiveram de ser feitas, uma vez que, para além de se tratarem de instruções relativamente rápidas, o resultado deve ser uniforme independentemente da rapidez dos ciclos do CPU. Por esta razão, em vez de serem utilizadas

medições do tempo de execução de uma instrução, foi utilizado um tempo de execução relativo, isto é, usou-se o ratio entre os tempos de execução de duas instruções, baseado no facto de podermos assumir que ambas executam igualmente lenta ou rapidamente num dado CPU.

Para evitar under e over measurement das tempos de execução foi medido o tempo de N execuções consecutivas de uma dada instrução, sendo este posteriormente dividido por N sendo que foram efectuadas várias medições desta forma e utilizado a menor. Para evitar mudanças de contexto e controlo de ciclos relativas a loops que pudessem poluir as medições, foram usadas estratégias de unrolling para um M de 100 e 10000, tendo-se decidido a favor do último devido à maior accuracy que oferecia. Por último, para tentar evitar problemas de cache, as primeiras medições foram descartadas e classificadas como um *"warm up"* fazendo com que, durante as medições utilizadas, o número de cache misses fosse reduzido ao mínimo, garantindo assim que todas as N instruções demorassem o mesmo tempo a ser executadas.

A instrução usada para medir o tempo de execução foi através da system call gettimeofday, que fornece time samples com precisão ao microssegundo, sendo eu houve a necessidade de calcular o overhead inerente à mesma, para que pudesse ser subtraído ao resultado obtido por forma a este ser corrigido. Finalmente, houve a necessidade de ter em atenção a escolha do N usado para as medições, sendo que um número reduzido de execuções consecutivas não nos dava um resultado representativo do tempo de execução (under measurements) mas um N elevado faria com que as interrupções geradas pelo hardware e geridas pelo sistema operativo poluíam em demasia as medições (noise measurements). Para tal, foi desenvolvido o script **suggest.c**, onde foi usado o método de medição a ser aplicado à detecção de hypervisors para medir o tempo de execução de uma instrução assembly *nop* (uma das mais rápidas, devido a fazer praticamente nada) para valores de N variáveis (Fig.1).

```
38      // warmup
39      for (int warmup=0; warmup <= range*10; warmup++) {
40          UNROLL_10000_TIMES(NOP);
41      }
42
43      int n = starting - step;
44      for (int i=0; n <= range; i++) {
45          long sampleTime;
46          long minInstructionTime = LONG_MAX;
47          n += step;
48
49          // performance measures for NOP instruction
50          for (int it=1; it <= C; it++) {
51              gettimeofday(&startTime, NULL);
52              for (int i=0; i < n/M; i++) {
53                  UNROLL_10000_TIMES(NOP);
54              }
55              gettimeofday(&endTime, NULL);
56
57              sampleTime = ((endTime.tv_sec * 1e6 + endTime.tv_usec) - (startTime.tv_sec * 1e6 + startTime.tv_usec));
58              if (sampleTime < minInstructionTime) {
59                  minInstructionTime = sampleTime;
60              }
61          }
62
63          results[i] = (minInstructionTime * 1e6) / n;
64          printf("\r[N = %d] (%.0f%%) - %ld ps\n", n, (i/((float) alloc))*100, results[i]);
65      }
66
```

Fig. 1. Snippet do script suggest.c, onde o principal loop de medições da instrução BASELINE, que representa uma instrução nop em assembly, efetuado com um N variante entre 'starting' e 'range', com um incremento de 'step'.

Estas medições foram executadas 100000 onde o menor valor obtido para cada N foi mantido e armazenado num ficheiro, ambos para um M de 100 e 10000 usado para o unroll. O script foi corrido múltiplas vezes para cada M e os resultados estão presentes em anexo (folha Suggest N do ficheiro HypervisorDetection.xslx). A partir destes dados, foram criados gráficos (Fig. 2) do tempo de execução em picosegundos em relação ao N usado em cada medição, onde a linha a sólido representa o mínimo de todas as medições executadas e as linhas a pontilhadas representam as medições com maiores desvios.

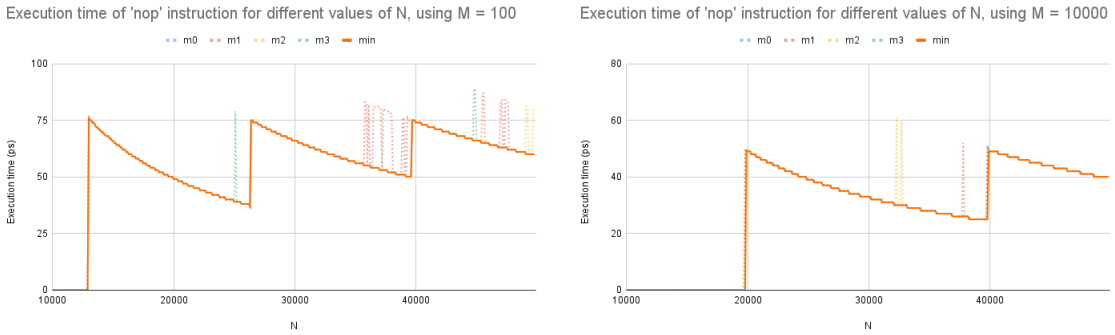

Fig. 2. Gráficos do tempos de execução (ps) em relação a N, para um unroll de M com valor 100 e 10000, respetivamente.

Apesar das medições com um M de valor 100 terem sido abandonadas numa segunda iteração da definição da estratégia, é interessante sublinhar que as suas medições geram muito mais interrupções quando comparadas ao M de 10000, o que era de esperar devido ao handling dos loops. Desta forma, o N escolhido para cada um dos casos foi 20000 e 30000 respectivamente, uma vez que ao analisar os gráficos vemos que estes valores estão relativamente perto do primeiro mínimo relativo maior que zero (que são causados por interrupções no hardware) enquanto que ainda deixam um pouco de espaço de manobra visto as medições podem não ser significativa no que toca a outliers.

Instruções utilizadas

Como tinha sido mencionado anteriormente, um tempo de execução absoluto não seria viável neste tipo de situações, sendo que a alternativa encontrada foi usar um tempo relativo sob a forma de um ratio entre o execution time da instrução em que temos interesse e o execution time de uma instrução baseline. Para a instrução baseline foi escolhida a instrução **nop**, devido à rapidez da sua execução e, para a instrução a ser medida, no caso de instruções assembly, existem várias CPU instructions que causam VM Exits, no entanto este trapping pode ser disabled quando se faz o setup da VM. Desta forma, existe uma lista de execuções da intel em que não é permitido este disable, mas grande parte delas são instruções privilegiadas, não sendo aplicáveis ao caso de detecção de hypervisors no ring 3, sendo por isso utilizado a instrução **CPUID**, que não é afetada por nenhuma destas restrições. O processo utilizado foi o já discutido acima, onde se medem N medições consecutivas com unroll, fazendo a diferença do ending time com o starting time e subtraindo-lhe o overhead da função gettimeofday, a medição da execução de uma função CPUID (macro para a utilização da instrução asm ‘cpuid’) pode ser vista no snippet de código da Fig. 3.

Quando a estratégia usando estava definida, foi então escolhida uma system call para substituir cpuid para comparar os resultados, tendo-se optado por **getpid**, que retorna informação do processo em si. A mesma lógica que se aplicou às CPU instructions aplica-se aqui, visto que existe uma trap quando se muda do ring 3 para o ring 0 na execução da mesma, que redireciona o código para o kernel, e caso esteja presente um hypervisor, quando se executam system calls rápidas, nota-se uma diferença nos tempos de execução enquanto esta trap é apanhada pelo hypervisor e redirecionada para o kernel processado pelo hypervisor.

Por último, quando se trata de um kernel module, já temos acesso às instruções privilegiadas que não podíamos usar no ring 0, então temos muitas mais opções. Foi tentado inicialmente usar a instrução **invd** mas, provavelmente devido a má utilização da mesma, provocava um crash no módulo, sendo que se optou por usar a instrução **out**.

```c

static float measure_asm_intruction(int iterations, int verbose) {
    struct timeval startTime;
    struct timeval endTime;
    struct timeval dummy;

    long sampleTime;
    long minInstructionTime = LONG_MAX;
    int regs[4];

    // warmups
    for (int warmup=0; warmup < WARMUPS; warmup++) {
        gettimeofday(&startTime, NULL);
        for (int i=0; i < N/M; i++) {
            UNROLL_10000_TIMES(CPUID);
        }
        gettimeofday(&endTime, NULL);
        sampleTime = ((endTime.tv_sec * 1e6 + endTime.tv_usec) - (startTime.tv_sec * 1e6 + startTime.tv_usec));
    }

    // performance measures for CPUID instruction
    for (int it=1; it <= iterations; it++) {
        gettimeofday(&startTime, NULL);
        for (int i=0; i < N/M; i++) {
            UNROLL_10000_TIMES(CPUID);
        }
        gettimeofday(&endTime, NULL);

        sampleTime = ((endTime.tv_sec * 1e6 + endTime.tv_usec) - (startTime.tv_sec * 1e6 + startTime.tv_usec));
        if (sampleTime < minInstructionTime) {
            minInstructionTime = sampleTime;
        }
    }

    return ((float) minInstructionTime * 1e3) / N;          // nanoseconds
}
```

Fig. 3. Snippet de detecteHypervisor.c, onde é feita a medição de N instruções CPUID consecutivas, executada várias iterações e mantido e devolvido o menor valor (em nanoseconds).

Detecção de hypervisors

Foi criado um módulo **detectHypervisor.c**, que contém métodos para medir a instrução baseline, o overhead, e as instruções mencionadas acima, assim como métodos que permitem detetar hypervisors tanto usando instruções assembly como usando system calls, que permite ajustes das threshold usadas através dos terminados em _customThreshold, sendo que nos restantes foi usado uma threshold pŕe-definida, obtida através de experimentações locais.

Para além do já mencionado **suggest.c**, foi também incluído um script **collect.c**, usado para colecionar dados para análise para o auxílio da definição de um threshold, assim como um script **detect.c** cujo objetivo é detectar e fazer o fingerprinting de um hypervisor, caso este exista.

É de fazer notar que, durante a migração da estratégia utilizado para o kernel module para se fazer detecção a nível do ring 0, houve umas pequenas alterações, nomeadamente no facto de mudar a system call utilizada para fazer medições do tempo com precisão aos nanosegundos, onde se passou a utilizar **getnstimeofday**, assim como alguns ajustes nas unidades de medidas para evitar uso de float points no módulo do kernel, sendo por isso as medições consideradas em picosegundos.

Testes e experiências locais

Por forma a se definir um threshold localmente, foram feitas experiências locais, tais como as descritas em [1], onde se fez uso dos métodos desenvolvidos no módulo detectHypervisors. A coleção de dados fez-se através da medição do ratio entre a instrução desejada e a baseline e esta medição foi

executada 1000 vezes e escolhida a menor. Este processo foi repetido 100 vezes, onde de cada uma se guardam os valores obtidos para os rations numa spreadsheet para futura análise.

Esta coleção de dados foi executada em sistemas nativos (n0-n2) assim como em sistemas virtualizados (v0-v2), cada um com um hypervisor diferente para se poderem comparar resultados e facilitar o processo de fingerprinting com base em thresholds. Estas experiências locais abrangem tanto coleção de dados usando assembly instructions e system calls no ring 3, assim como instruções privilegiadas em assembly no ring 0, sendo para todas usada o M escolhido acima. A única exceção foi a primeira tentativa nesta abordagem em que se usou um M de 100 para assembly instructions no user space, que não vai ser discutidas mas foi incluída nos resultados de qualquer forma, tendo-se repetido a experiência posteriormente para o M correto.

Os resultados estão presentes no anexo **HypervisorDetection.xslx** (Folha 2 a 5) e a partir destes foram feitos *box graphs* para cada tipo de detecção, onde as boxes mostram o intervalo de 25 a 75% dos dados e os whiskers indicam intervalos de 5-95%, exceto no caso das system calls em que este último intervalo é modificado para 10-90% devido a haver valores com menor conformidade com os restantes nos extremos que dificultavam o processo de escolha de um threshold. Foi a partir destes gráficos que se definiram os thresholds usados para a detecção, que estão representados nos mesmos pela linha vermelha a pontilhado.

Resultados, thresholds e fingerprinting

Os gráficos da Fig. 4 representam os resultados obtidos a partir das medições feitas no user space, para instruções assembly usando o M de 100 e 10000, e para as system calls usando um M de 10000, respectivamente. Por observação dos mesmos, pode-se concluir que a presença de hypervisors é muito mais clara quando se usam assembly instructions, visto que o ratio entre tempos de execução têm um gap muito maior quando se trata de sistemas nativos e sistemas virtualizados. Neste caso o threshold é relativamente fácil de definir, ao contrário do caso das system calls, onde se tiveram de fazer mais medições (e até reduzir a percentagem de dados a considerar).

Além disso, consegue-se também perceber que existe uma diferença significativa entre o ratio de tempos de execução entre diferentes tipos de hypervisors, onde o primeiro (v0, que corre Virtualbox) é quase o dobro do segundo (v1, onde se corre QEMU). Os menores valores obtidos para o ratio foram obtidos no sistema virtualizado v2, a correr VMWare, não havendo uma diferença tão grande comparado com o v1. No entanto, esta diferença é o que nos permite fazer o fingerprinting de hypervisors e, segundo estes resultados, foram impostos threshold para cada um dos hypervisors analisados, isto é, foi-se definido um limiar máximo e mínimo para cada e, se as medições executadas estiverem entre essa área, então assumimos ser esse tipo de hypervisor. Esta divisão de hypervisors foi efetuada para o método onde aplicavam assembly instructions apenas, devido ao facto de que, no método em que se usam system calls, a linha que separava os diferentes hypervisors é tão tênues que torna difícil a definição confiante de thresholds assim como a obtenção de uma taxa de falsos-positivos/falsos.negativos decente.

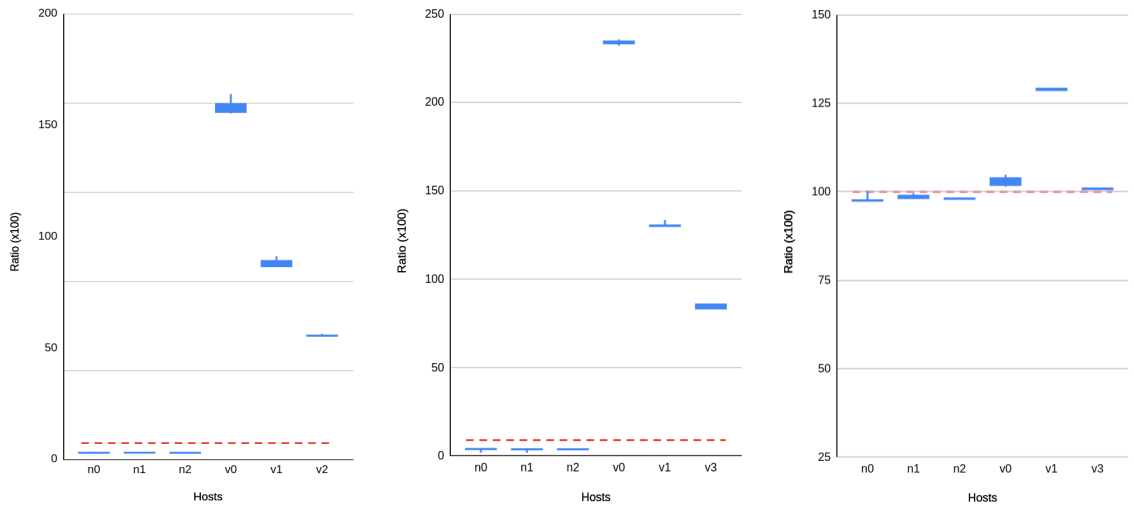

Fig. 4. Box graphs representando a dispersão de ratios entre os tempos de execução, para hosts nativos (n0-n2 sempre no mesmo host) e para hosts virtualizados (onde v0 corre VirtualBox, v1 corre QEMU e v2 corre VMWare). No primeiro caso foi usado um M=100 e assembly instructions , no segundo foi usado um M = 10000 e assembly instructions e no último foi usado um M = 10000 e system calls

É de fazer notar que o threshold usado para diferenciar entre sistemas virtualizados e sistemas nativos pode ser ajustado conforme a taxa de falsos positivos ou a taxa de falsos negativos que alguém precise ou esteja disposto a abdicar. Caso se trate de implementação desta detecção em malwares, é possível diminuir a taxa de falsos negativos, para garantir que não existem virtualizadores a serem detectados como sistemas nativos, podendo resultar em perda de efetividade mas não se arriscando a reverse engineering usando honey pots, por exemplo. O mesmo pode ser dito para diminuir a taxa de falsos positivos, se alguém estiver pronto a abdicar a alguns sistemas nativos serem classificados como virtualizadores, dependendo do uso e função do mesmo.

Limitações e conclusões

Os métodos implementados provaram-se eficientes, sendo que alguns mais que outros, visto que os métodos que usam system calls tornam difícil a divisão entre hypervisors diferentes, contudo continuando a ser viáveis quando o objetivo é separar sistemas nativos de sistemas virtualizados. No entanto, não se pode deixar de frisar que este método não é o mais stealthy possível devido ao número de vezes que as instruções cpuid e getpid são feitas que, apesar de não serem propriamente perigosas ou pouco comuns, o elevado número de chamadas feitas num tão curto espaço de tempo pode ser visto como uma red flag.

Por último, e admissivelmente, a estratégia implementada usando kernel modules não foi tão explorada como gostaria de ter sido, uma vez que o elevado tempo de execução tornou difícil a recolha de dados que pudessem realmente refletir os tempos de execução em native hosts, muito menos em sistemas virtualizados. No entanto, e assumindo que seguiria o padrão da estratégia aplicadas ao user space, usando assembly instructions, o threshold foi mais fácil de identificar com um número de medições muito menores.

Referências

[1] Brengel, M., Backes, M., Rossow, C. (2016). Detecting Hardware-Assisted Virtualization. In: Caballero, J., Zurutuza, U., Rodríguez, R. (eds) Detection of Intrusions and Malware, and Vulnerability Assessment. DIMVA 2016. Lecture Notes in Computer Science(), vol 9721. Springer, Cham. https://doi.org/10.1007/978-3-319-40667-1_11

[2] Korkin, Igor. (2015). Two Challenges of Stealthy Hypervisors Detection: Time Cheating and Data Fluctuations. https://www.researchgate.net/publication/278241657_Two_Challenges_of_Stealthy_Hypervisors_Detection_Time_Cheating_and_Data_Fluctuations