Load+Reload: Uma prova de conceito de um side-channel que explora a cache associativa de processadores AMD

Gabriel da Fonseca Ottoboni Pinho

10 de fevereiro de 2022

1 Introdução

Em busca da maior performance possível, os processadores atuais tiram vantagem de diversos artifícios, muitas vezes sem grande consideração pela segurança. Nesse contexto, ataques como Spectre [3] e Meltdown [5] mostram que a execução especulativa é um tópico muito interessante a ser explorado. Nesse artigo, construiremos uma prova de conceito do ataque Load+Reload [6], que tira vantagem do preditor de *way* presente na cache dos processadores atuais da AMD. Durante esse processo, veremos uma forma diferente de medir o tempo de leitura da memória, a thread contadora, mais precisa do que a instrução `rdtsc`, usada em ataques como Flush+Reload [7].

2 Organização interna da cache

A cache de um processador é uma memória que armazena páginas de memória recentemente acessadas. Por na maior parte das vezes ser composta por células de memória estáticas (SRAM), a cache é múltiplas vezes mais rápida que a memória principal de um computador, que usa células dinâmicas (DRAM).

A menor unidade dentro da cache é chamada de **linha** e geralmente tem o mesmo tamanho de uma página de memória. Uma certa página, no entanto, não necessariamente pode ser armazenada em qualquer linha. O grupo de linhas nas quais uma página pode ser amazenada é chamada de **set**. Além disso, cada linha de um set onde uma certa página pode ser armazenada é chamada de **way**.

No caso mais simples, cada set tem apenas 1 way e a cache é considerada **diretamente mapeada**, pois cada página só pode ser armazenada em exatamente uma linha. Se, pelo contrário, uma página pode ser armazenada em *qualquer* linha, a cache é considerada **totalmente associativa**, como se tivesse apenas 1 grande set. Por fim, se uma página pode ser armazenada em n linhas, a cache é considerada **associativa por conjunto n-way**.

Outro ponto importante é a forma como a cache será indexada e marcada. O **index** se refere ao endereço utilizado para acessar um certo set, enquanto que a **marca**[1] se refere a como cada way dentro de um set será identificado. Nos processadores Zen da AMD, a cache é **virtualmente indexada e fisicamente marcada (VIPT)**[2][1]. Isso significa que a partir do endereço virtual de uma página é possível encontrar o set onde ela está armazenada na cache. Analogamente, a partir do endereço físico de uma página é possível encontrar em qual way do set página está.

3 Predição de way

Como explicado no tópico anterior, quando uma página de memória é requisitada, primeiro o endereço virtual dela é utilizado para determinar o set da cache onde ela pode estar armazenada. Após determinar o set, o endereço físico é utilizado para encontrar o way. A grande vantagem desse método é que o endereço físico pode ser calculado enquanto o set está sendo determinado.

Na arquitetura Zen da AMD, a cache L1 de dados é associativa por conjunto 8-way[1], ou seja, pode ser necessário verificar a marca de 8 ways para saber em qual deles uma página pode estar armazenada. A fim de acelerar esse processo, o processador usa uma μmarca para determinar se a página está ou não em um set[1]. A μmarca é uma função do endereço virtual e, analogamente à marca normal, fica armazenada em cada way. Como ela não depende do endereço físico, o processador pode rapidamente verificar se algum way tem a μmarca correspondente e, se nenhum tiver, imediamente desistir da cache L1 e começar a procurar a página na cache L2, sem nem mesmo esperar a tradução do endereço virtual para o físico.

O problema ocorre quando dois endereços virtuais *distintos* referentes ao *mesmo* endereço físico são acessados em sucessão. Cada acesso vai alterar a μmarca, fazendo com que o próximo acesso caia para a cache L2, já que o preditor de way não encontrará a μmarca que foi sobrescrita[1]. Nesse caso, todos os acessos serão feitos na cache L2, aumentando o tempo de acesso.

4 Ataques de cache

Um side-channel se baseia em extrair informação por meio de algum rastro deixado pela implementação de um sistema. A cache de um processador, por exemplo, implicitamente informa um processo se uma certa página de memória foi acessada recentemente.

Se dois processos A e B compartilham a mesma página de memória, o processo B pode medir o tempo gasto para ler a página e determinar se a mesma está ou não na cache. Caso a leitura seja rápida, a página estava na cache; caso contrário, não estava. Dessa forma, se o processo B sabe que a página não estava na cache em um momento t_1 e em um momento posterior t_2 for determinado que a página estava na cache, pode-se concluir que o processo A a acessou.

[1] *Tag* em inglês.

[2] *Virtually indexed and physically tagged (VIPT)* em inglês.

5 Medindo o tempo de acesso

5.1 `rdtsc`

No artigo original do ataque Flush+Reload [7], a instrução `rdtsc` é utilizada para pedir o tempo de acesso à memória e determinar se uma certa página está ou não na cache. Essa instrução carrega o valor do *time-stamp counter* nos registradores `edx:eax`. Assim, em conjunto com instruções de ordenação de execução, é possível calcular o tempo decorrido entre o início e fim da leitura.

```
uint64_t load_count(uint64_t *addr)
{
    uint64_t volatile time;
    asm volatile (
        "mfence              \n\t"
        "lfence              \n\t"
        "rdtsc               \n\t"
        "lfence              \n\t"
        "movl %%eax, %%ebx   \n\t"
        "movq (%%rcx), %%rcx \n\t"
        "lfence              \n\t"
        "rdtsc               \n\t"
        "subl %%ebx, %%eax   \n\t"
        : "=a" (time)
        : "c" (addr)
        : "rbx"
    );
    return time;
}
```

Figura 1: Função que calcula o tempo de acesso a uma página usando a instrução `rdtsc`.

Um problema ao usar a instrução `rdtsc` é que nos processadores mais recentes da AMD ela não é mais tão precisa, pois tem seu contador incrementado a cada 30 ciclos aproximadamente [6]. Esse fato diminui muito a resolução da medição, principalmente em ataques nos quais a diferença do tempo de acesso é mais sutil.

5.2 A thread contadora

```
uint64_t volatile count = 0;
void *counting_thread(void *args) {
    set_affinity(COUNTING_CORE);
    asm volatile (
        "xorq %%rax, %%rax   \n\t"
        "loop%=:             \n\t"
        "incq %%rax          \n\t"
        "movq %%rax, (%%rbx) \n\t"
        "jmp loop%=          \n\t"
        :
        : "b" (&count)
        : "rax"
    );

    pthread_exit(NULL);
}
```

Figura 2: Função da thread contadora.

O artigo ARMageddon [4] mostrou que threads contadoras podem ter uma resolução tão boa ou melhor do que a instrução `rdtsc` em processadores ARM. A ideia é ter uma thread cujo único objetivo é incrementar uma

variável global continuamente o mais rápido possível.

A fim de ser o mais otimizada possível, a thread contadora armazena o valor atual de `count` no registrador `rax` e o copia para a memória após incrementá-la. Dessa forma, apenas uma escrita à memória é realizada e o valor atual é acessado exclusivamente pelo registrador.

Um ponto importante a ser notado no código da thread contadora é o uso das instruções de ordenação `mfence` e `lfence`. Analogamente à medição do tempo usando a instrução `rdtsc`, é necessário garantir que as duas leituras à variável `count` aconteçam antes e depois da leitura da página. Além disso, para o caso da thread contadora, é importante que a thread principal do atacante obtenha um valor minimamente atualizado de `count`. A seguir está reproduzido um trecho do manual da Intel sobre a instrução `mfence`.

> This serializing operation guarantees that every load and store instruction that precedes the MFENCE instruction in program order becomes globally visible before any load or store instruction that follows the MFENCE instruction. ([2])

Dessa forma, segundo o manual, os incrementos realizados pela thread contadora na variável `count` durante a leitura da página se tornarão visíveis globalmente antes da segunda leitura de `count` que segue a instrução `mfence`. O uso dessa instrução é fundamental, pois sem ela as duas leituras de `count` retornariam o mesmo valor na maior parte das vezes, o que resultaria em um tempo de leitura nulo.

```
uint64_t load_count(uint64_t *addr)
{
    uint64_t volatile time;
    asm volatile (
        "mfence              \n\t"
        "lfence              \n\t"
        "movq (%%rbx), %%rcx \n\t"
        "lfence              \n\t"
        "movq (%%rax), %%rdx \n\t"
        "lfence              \n\t"
        "mfence              \n\t"
        "movq (%%rbx), %%rax \n\t"
        "subq %%rcx, %%rax   \n\t"
        : "=a" (time)
        : "a" (addr), "b" (&count)
        : "rcx", "rdx"
    );
    return time;
}
```

Figura 3: Função que calcula o tempo de acesso a uma página usando a thread contadora.

6 Load+Reload

Load+Reload é um dos dois ataques propostos no artigo Take a Way[6] e tira vantagem do fato que quando dois endereços virtuais *distintos* referentes ao *mesmo* endereço físico são acessados em sucessão, a leitura ocorrerá na cache L2. Esse comportamento, entretanto,

```
read_byte(&data[secret[i] * 4096]);
```

Figura 4: Leitura do arquivo a partir da string `secret`.

só ocorre quando as threads sendo executadas estão no mesmo núcleo físico[6].

Foram então criados dois programas, uma vítima e um atacante, que serão executados em núcleos lógicos distintos, mas no mesmo núcleo físico. Para isso, foi utilizada a biblioteca `pthread` para criar a thread e prendê-la a um núcleo específico.

A memória que será compartilhada entre a vítima e o atacante é um arquivo de 8MiB composto de bytes aleatórios mapeados na memória com a syscall `nmap`. O ataque consiste em uma vítima lendo continuamente partes do arquivo `data` usando caracteres de uma string `secret`. O caractere é multiplicado por 4096, de modo que cada byte do arquivo lido está em uma página distinta.

Enquanto isso, o atacante também lê continuamente o arquivo. Diferente da vítima, no entanto, o atacante selectiona todos os bytes de `0` até `0xff` como índice para a leitura do arquivo.

```
uint64_t time =
    load_count(&data[byte * 4096]);
```

O tempo levado para a leitura é então medido usando uma thread contadora e o byte que levou mais tempo para ser lido é escolhido como o que a vítima estava lendo naquele momento.

A ideia é que se o byte sendo lido pelo atacante for o mesmo que está sendo lido pela vítima, a leitura do atacante cairá para a L2 por conta do preditor de way. Essa leitura na L2 levará consideravelmente mais tempo do que uma leitura na L1, e essa diferença pode ser medida usando a thread contadora.

Referências

[1] AMD. *Software Optimization Guide for AMD Family 17h Models 30h and Greater Processors*. 2019. URL: https://developer.amd.com/wp-content/resources/56305_SOG_3.00_PUB.pdf.

[2] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 2021. URL: https://cdrdv2.intel.com/v1/dl/getContent/671200.

[3] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". Em: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 1–19. DOI: 10.1109/SP.2019.00002.

[4] Moritz Lipp et al. "ARMageddon: Cache Attacks on Mobile Devices". Em: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, ago. de 2016, pp. 549–564. ISBN: 978-1-931971-32-4. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lipp.

[5] Moritz Lipp et al. "Meltdown: Reading Kernel Memory from User Space". Em: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, ago. de 2018, pp. 973–990. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

[6] Moritz Lipp et al. "Take A Way: Exploring the Security Implications of AMD's Cache Way Predictors". Em: *15th ACM ASIA Conference on Computer and Communications Security (ACM ASIACCS 2020)*. Taipei, Taiwan, 2020. DOI: 10.1145/3320269.3384746. URL: https://hal.inria.fr/hal-02866777.

[7] Yuval Yarom e Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". Em: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, ago. de 2014, pp. 719–732. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.