Meltdown and Spectre Samples

Written in Assembly

U. Plonus

March 10, 2018

Contents

1 Introduction

1.1 Overview

TBD

1.2 Nasm

TBD

5 ⟨*preamble* 5⟩≡ (9f)

```
bits 64
      global          _start
      pagesize        equ  4096
```

Defines:
`_start`, used in chunk 9a.
`pagesize`, used in chunks 8b and 9c.

2 Cache Access Timing

2.1 Introduction

TBD

2.2 Detect Cache Access Time

2.2.1 High Resolution Timer

First we need a high resolution timer to determine the cache access time. For this Intel® processors since the Pentium® model have a time stamp counter. The time stamp counter is monotonically incrementing. When reading the time stamp counter (with `rdtsc`) the result is delivered back in the registers `EDX` and `EAX` forming a 64bit value. The time stamp counter is not an absolute value but a relative value, meaning that you cannot (easily) calculate from the time stamp counter to some time units (e.g. ns). But this is no problem as we only want to measure relative times.

To retrieve a 64bit value for the time we shift the value in `EDX` 32 bits to the left and add the value of `EAX` to this.

7 ⟨*tsc-64bit* 7⟩≡ (8a)

```
rdtsc
shl     RDX,32
add     RAX,RDX
```

2.2.2 Cache Access Time Routine

Next we need a routine that calculates the cache access time for us.

First we have to ensure in this routine that the speculative execution of the processor does not interfere with our time measurement. For this we use the instruction `lfence` which ensures that all previous reads are done before executing the next instructions.

Next we access a memory location with the address `RDI` by loading this into `RCX` and measure the time before and after the access.

The command `lfence` before reading the time stamp counter is needed because we have to ensure that all reads before the time measurements are done.

At last we calculate the relative time needed to access the memory location. In theory we should see a difference whether the memory location is accessed before or not.

Parameters

`RDI` the address of the memory which is loaded either from the cache or from memory

8a ⟨*calculate-cache-access-time* 8a⟩≡ (10a)

```
_calccachetime:
        lfence
  ⟨tsc-64bit 7⟩
        mov     R8,RAX
        mov     RCX,[RDI]
        lfence
  ⟨tsc-64bit 7⟩
        sub     RAX,R8
        ret
```

Defines:
`_calccachetime`, never used.

2.3 Measure Cache Access Time

To measure the cache timing we create a standalone program that shows us the time for a cached and for an uncached memory access.

First we need some area in memory with data which we can later read from. This data area goes into the area `.bss` which contains uninitialized data. We align the data at a page boundary and reserve 2 pages for our data.

8b ⟨*cachetiming-uninitialized-data* 8b⟩≡ (9f)

```
section .bss
        align           pagesize
        data:           times 2 resb pagesize
```

Defines:
`data`, used in chunk 9b.
Uses `pagesize` 5.

The program begins with the label `_start` and is in the section `.text`.

9a ⟨*cachetiming-program* 9a⟩≡ (9f) 9b ▷

```
section .text
_start:
```

Uses `_start` 5.

Now we start with initialising the `data` area with some random data. For this we load `RDI` with the address of the `data` area.

9b ⟨*cachetiming-program* 9a⟩+≡ (9f) ◁ 9a 9c ▷

```
        mov        RDI,data
```

Uses `data` 8b.

Next we load the number of bytes to fill into `RSI`. For this we load the `pagesize` into `RSI` and multiply it with 2 by shifting the value 1 bit to the left.

9c ⟨*cachetiming-program* 9a⟩+≡ (9f) ◁ 9b 9d ▷

```
        mov        RSI,pagesize
        shl        RSI,1
```

Uses `pagesize` 5.

At last we load `EDX` with some random seed. For this we use `rdtsc` and only use the lower 32 bit of the value.

9d ⟨*cachetiming-program* 9a⟩+≡ (9f) ◁ 9c 9e ▷

```
        rdtsc
        mov        EDX,EAX
```

Now we call `_xorshift` to fill the `data` area.

9e ⟨*cachetiming-program* 9a⟩+≡ (9f) ◁ 9d 10a ▷

```
        call       _xorshift
```

Uses `_xorshift` 14a.

TBD

9f ⟨*cachetiming.asm* 9f⟩≡

⟨*preamble* 5⟩

⟨*cachetiming-rodata* 10b⟩

⟨*cachetiming-uninitialized-data* 8b⟩

⟨*cachetiming-program* 9a⟩

10a ⟨*cachetiming-program* 9a⟩+≡ (9f) ◁9e

```
        mov     RDI,scached
        call    _print
        mov     RDI,scr
        mov     RSI,1
        call    _nprint
        mov     RDI,suncached
        call    _print
        mov     RDI,scr
        mov     RSI,1
        call    _nprint
  ⟨exitProgram 13b⟩

  ⟨calculate-cache-access-time 8a⟩

  ⟨xorshift-prng 14a⟩

  ⟨utilities 13a⟩
```

Uses `_nprint` 15a, `_print` 15b, `scached` 10b, `scr` 10b, and `suncached` 10b.

TBD

10b ⟨*cachetiming-rodata* 10b⟩≡ (9f)

```
  section .rodata
        suncached:      db "Uncached Access Time: ",0x00
        scached:        db "Cached Access Time: ",0x00
        scr:            db 0x0a
```

Defines:
`scached`, used in chunk 10a.
`scr`, used in chunk 10a.
`suncached`, used in chunk 10a.

2.4 Read Array via Cache Access Time

TBD

3 Signals

3.1 Basics

TBD

3.2 Detecting Signals

TBD

3.3 Handling Signals

TBD

4 Utilities

4.1 Introduction

TBD

13a ⟨*utilities* 13a⟩≡ (10a)

⟨*nprint* 15a⟩

⟨*print* 15b⟩

4.2 Exit Program

TBD

13b ⟨*exitProgram* 13b⟩≡ (10a)

```
        xor       RDI,RDI
        mov       RAX,60
        syscall
```

4.3 Random Number Generator

To initialize the data a random number generator (RNG) is used. The sample programs use xorshift[1] as RNG.

First we move the number of values to be generated to `RCX` (which is a counter in x86 processors) and divide it by 4 (because we use a 32bit RNG). Additionally we move the seed to `EAX`.

Parameters

`RDI` the address of the memory which is to be filled with random numbers

`RSI` the number of bytes that are filled with random numbers. This must be a multiple of 4

`EDX` the seed of the RNG

14a ⟨*xorshift-prng* 14a⟩≡ (10a) 14b▷

```
_xorshift:
      mov       RCX,RSI
      shr       RCX,2
      mov       EAX,EDX
```

Defines:
`_xorshift`, used in chunk 9e.

Now we can generate the next 32bit random number.

14b ⟨*xorshift-prng* 14a⟩+≡ (10a) ◁14a 14c▷

```
.next_random:
      mov       EBX,EAX
      shl       EAX,13
      xor       EAX,EBX
      mov       EBX,EAX
      shr       EAX,17
      xor       EAX,EBX
      mov       EBX,EAX
      shl       EAX,5
      xor       EAX,EBX
```

Because we want to generate multiple random numbers we store the value of `EAX` to `[RDI]` and loop for the next random number.

14c ⟨*xorshift-prng* 14a⟩+≡ (10a) ◁14b

```
      stosd
      loop      .next_random
      ret
```

[1] https://en.wikipedia.org/wiki/Xorshift

4.4 Printing Strings

4.4.1 Printing Strings with Length

The routine _nprint prints a string with the given length to stdout.

We move the number of bytes to print to RDX which is the 3rd parameter to the systemcall. Next we move the address of the bytes to print to RSI which is the 2nd parameter to the systemcall. The 1st argument (in RDI) to the systemcall is the file descriptor (1 is stdout). Additionally the number of the systemcall (1) is passed in RAX. The systemcall (syscall) now prints RDX bytes from [RSI] to the file descriptor RDI.

At the end we return to the caller.

Parameters

RDI the number of bytes to print to stdout

RSI the address to the bytes to print to stdout

15a ⟨*nprint* 15a⟩≡ (13a)

```
_nprint:
      mov     RDX,RSI
      mov     RSI,RDI
      mov     RDI,1
      mov     RAX,1
      syscall
      ret
```

Defines:
_nprint, used in chunks 10a and 16c.

4.4.2 Printing C-Strings

The routine _print prints a null-terminated string to stdout.

So first we start with clearing AL (setting it to null) and saving the address of the string to RSI. We're using RSI because we later need the address to calculate the length of the string.

Parameters

RDI the address to the null-terminated bytes to print to stdout

15b ⟨*print* 15b⟩≡ (13a) 16a ▷

```
_print:
      xor     AL,AL
      mov     RSI,RDI
```

Defines:
_print, used in chunk 10a.

Next we search for the terminating `null` ('\0') character. For this we use the instruction `scasb` (scan string byte) which compares the byte at the address `[RDI]` with the value in `AL` and sets the flags accordingly. When the byte at `[RDI]` is not the value of `AL` the next instruction (`jne`) jumps to the given label (`.next_char` in this case).

`scasb` additionally increments `RDI` so that we go through the string until '\0' is found.

16a ⟨*print* 15b⟩+≡ (13a) ◁15b 16b▷

```
.next_char:
      scasb
      jne        .next_char
```

After we have found the string termination we calculate the number of bytes that the string has. For this we exchange the registers `RDI` and `RSI`. In `RDI` we now have the starting address of the bytes to print and in `RSI` we have the end address of the bytes to print. After that we calculate the number of bytes to print.

16b ⟨*print* 15b⟩+≡ (13a) ◁16a 16c▷

```
      xchg       RDI,RSI
      sub        RSI,RDI
```

Now we have the address of the string in `RDI` and the length of the string in `RSI` which are the 1st and 2nd argument in the call of `_nprint`.

16c ⟨*print* 15b⟩+≡ (13a) ◁16b

```
      call       _nprint
      ret
```

Uses `_nprint` 15a.

4.5 Printing Numbers

TBD

A Glossary

x86 x86 denotes a microprocessor architecture based on the 8086/8088 14

B Acronyms

RNG random number generator 14

C x86-Instructions

D Code Chunks