Meltdown and Spectre Samples

Written in Assembly

U. Plonus

March 10, 2018

Contents

1 Introduction **5**
- 1.1 Overview 5
- 1.2 Nasm 5

2 Cache Access Timing **7**
- 2.1 Introduction 7
- 2.2 Detect Cache Access Time 7
  - 2.2.1 High Resolution Timer 7
  - 2.2.2 Cache Access Time Routine 8
- 2.3 Measure Cache Access Time 9
  - 2.3.1 Setup 9
  - 2.3.2 Measure Time 10
- 2.4 Read Array via Cache Access Time 11

3 Signals **13**
- 3.1 Basics 13
- 3.2 Detecting Signals 13
- 3.3 Handling Signals 13

4 Utilities **15**
- 4.1 Introduction 15
- 4.2 Exit Program 15
- 4.3 Random Number Generator 16
- 4.4 Printing Strings 17
  - 4.4.1 Printing Strings with Length 17
  - 4.4.2 Printing C-Strings 17
- 4.5 Printing Numbers 18

A Glossary **19**

B Acronyms **21**

C x86-Instructions **23**

D Code Chunks **25**

1 Introduction

1.1 Overview

TBD

1.2 Nasm

TBD

5 ⟨*preamble* 5⟩≡ (10d)

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

Defines:
`_start`, used in chunk 9b.
`pagesize`, used in chunk 9.

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 we use the 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⟩≡ (8)

```
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

8 ⟨*calculate-cache-access-time* 8⟩≡ (11a)

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

Defines:
`_calccachetime`, used in chunk 10c.

2.3 Measure Cache Access Time

2.3.1 Setup

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.

9a ⟨*cachetiming-uninitialized-data* 9a⟩≡ (10d)

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

Defines:
`data`, used in chunks 9c and 10a.
Uses `pagesize` 5.

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

9b ⟨*cachetiming-program* 9b⟩≡ (10d) 9c ▷

```
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.

9c ⟨*cachetiming-program* 9b⟩+≡ (10d) ◁ 9b 9d ▷

```
      mov       RDI,data
```

Uses `data` 9a.

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.

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

```
      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.

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

```
      rdtsc
      mov       EDX,EAX
```

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

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

```
      call      _xorshift
```

Uses `_xorshift` 16a.

2.3.2 Measure Time

Now that we have setup our `data` area we can now cache data from the first page by loading it into a register which also loads this into the cache.

For this we load `RDI` with the address of the `data` area.

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

```
        mov         RDI,data
```

Uses `data` 9a.

Before we load the data into a register now we will clear the cache lines with the given address. For this we use the instruction `clflush`. After that we will load the data into a register.

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

```
        clflush     [RDI]
        mov         RCX,[RDI]
```

Now we can determine the time that is needed to load this data once again. We do not need to load `RDI` again because it has not changed.

10c ⟨*cachetiming-program* 9b⟩+≡ (10d) ◁10b 11a▷

```
        call        _calccachetime
```

Uses `_calccachetime` 8.

Now we have the relative cache access time in register `RAX`. Now we can print the measured time to `stdout`.

TBD

10d ⟨*cachetiming.asm* 10d⟩≡

```
  ⟨preamble 5⟩

  ⟨cachetiming-rodata 11b⟩

  ⟨cachetiming-uninitialized-data 9a⟩

  ⟨cachetiming-program 9b⟩
```

11a ⟨*cachetiming-program* 9b⟩+≡ (10d) ◁10c

```
        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* 15b⟩

⟨*calculate-cache-access-time* 8⟩

⟨*xorshift-prng* 16a⟩

⟨*utilities* 15a⟩

Uses `_nprint` 17b, `_print` 18a, `scached` 11b, `scr` 11b, and `suncached` 11b.

TBD

11b ⟨*cachetiming-rodata* 11b⟩≡ (10d)

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

Defines:
`scached`, used in chunk 11a.
`scr`, used in chunk 11a.
`suncached`, used in chunk 11a.

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

15a ⟨*utilities* 15a⟩≡ (11a)

⟨*nprint* 17b⟩

⟨*print* 18a⟩

4.2 Exit Program

TBD

15b ⟨*exitProgram* 15b⟩≡ (11a)

```
        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 clear the direction flag to ensure that we are incrementing the data pointer `RDI`.

Next 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

16a ⟨*xorshift-prng* 16a⟩≡ (11a) 16b▷

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

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

Now we can generate the next 32bit random number.

16b ⟨*xorshift-prng* 16a⟩+≡ (11a) ◁16a 17a▷

```
.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
```

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

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

17a ⟨*xorshift-prng* 16a⟩+≡ (11a) ◁16b

```
        stosd
        loop      .next_random
        ret
```

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

17b ⟨*nprint* 17b⟩≡ (15a)

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

Defines:
_nprint, used in chunks 11a and 18d.

4.4.2 Printing C-Strings

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

First we clear the direction flag to increment the address in RDI while scanning the data.

Next 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`

18a ⟨*print* 18a⟩≡ (15a) 18b▷
```
_print:
      cld
      xor       AL,AL
      mov       RSI,RDI
```
Defines:
`_print`, used in chunk 11a.

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.

18b ⟨*print* 18a⟩+≡ (15a) ◁18a 18c▷
```
.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.

18c ⟨*print* 18a⟩+≡ (15a) ◁18b 18d▷
```
      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`.

18d ⟨*print* 18a⟩+≡ (15a) ◁18c
```
      call      _nprint
      ret
```
Uses `_nprint` 17b.

4.5 Printing Numbers

TBD

A Glossary

x86 x86 denotes a microprocessor architecture based on the 8086/8088 16

B Acronyms

RNG random number generator 16

C x86-Instructions

`clflush` Flush Cache Line, introduced with Intel® Pentium® 4 10

`lfence` Load Fence, introduced with Intel® Pentium® 4 8

`rdtsc` Read Time Stamp Counter, introduced with Intel® Pentium® 7, 9

D Code Chunks

⟨*cachetiming-program* 9b⟩
⟨*cachetiming-rodata* 11b⟩
⟨*cachetiming-uninitialized-data* 9a⟩
⟨*cachetiming.asm* 10d⟩
⟨*calculate-cache-access-time* 8⟩
⟨*exitProgram* 15b⟩
⟨*nprint* 17b⟩
⟨*preamble* 5⟩
⟨*print* 18a⟩
⟨*tsc-64bit* 7⟩
⟨*utilities* 15a⟩
⟨*xorshift-prng* 16a⟩