Meltdown and Spectre Samples

Written in Assembly

U. Plonus

March 12, 2018

Contents

1 Introduction

1.1 Overview

TBD

1.2 Nasm

TBD

5 ⟨*preamble* 5⟩≡ (12a 13b)

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

Defines:
`_start`, used in chunks 9c and 13a.
`pagesize`, used in chunks 8b, 9e, and 12c.

1.3 Conventions

In this section we define some convention for names that are specific for this document.

The data is divided into three parts: read-only data, initialized data and uninitialized data. Code chunks with this type of data all have a defined suffix.

Definition 1 *Read-only data is data that is not modified during program execution. The suffix for read-only data is **-rodata**.*

Definition 2 *Initialized data is data that is changeable during program execution. The data is already initialized with data when the program starts. The suffix for initialized data is **-idata**.*

Definition 3 *Uninitialized data is data that is changeable during program execution. The data is not initialized. The suffix for uninitialized data is **-udata**.*

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⟩≡ (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⟩≡ (12a 13b)

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

Defines:
`_calccachetime`, used in chunks 10b and 11a.

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.

8b ⟨*cachetiming-udata* 8b⟩≡ (12a) 9b ▷

```
        align       pagesize
        data:       times 2 resb pagesize
```

Defines:
`data`, used in chunks 9, 11a, and 12c.
Uses `pagesize` 5.

From time to time we need a small scratch area so we define an area with 32 bytes.

9a ⟨*scratch-data* 9a⟩≡ (9b 12d)
```
        scratch:        resb 32
```
Defines:
`scratch`, used in chunks 10e and 11c.

9b ⟨*cachetiming-udata* 8b⟩+≡ (12a) ◁8b
⟨*scratch-data* 9a⟩

The program begins with the label `_start`.

9c ⟨*cachetiming-program* 9c⟩≡ (12a) 9d▷
```
_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.

9d ⟨*cachetiming-program* 9c⟩+≡ (12a) ◁9c 9e▷
```
        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.

9e ⟨*cachetiming-program* 9c⟩+≡ (12a) ◁9d 9f▷
```
        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.

9f ⟨*cachetiming-program* 9c⟩+≡ (12a) ◁9e 9g▷
```
        rdtsc
        mov         EDX,EAX
```

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

9g ⟨*cachetiming-program* 9c⟩+≡ (12a) ◁9f 9h▷
```
        call        _xorshift
```
Uses `_xorshift` 18a.

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.

9h ⟨*cachetiming-program* 9c⟩+≡ (12a) ◁9g 10a▷
```
        mov         RDI,data
```
Uses `data` 8b.

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 flushing the cache line we ensure (with `lfence`) that all reads from memory are finished before we load the data into a register again (and filling the cache).

10a ⟨*cachetiming-program* 9c⟩+≡ (12a) ◁9h 10b▷

```
    clflush   [RDI]
    lfence
    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.

10b ⟨*cachetiming-program* 9c⟩+≡ (12a) ◁10a 10d▷

```
    call      _calccachetime
```

Uses `_calccachetime` 8a.

Now we have the relative cache access time in register `RAX`. We store this value to the stack and print out an explaining text.

For this we define the text to print and (as a helper) a carriage return (CR).

10c ⟨*cachetiming-rodata* 10c⟩≡ (12a) 11b▷

```
    scr:          db 0x0a
    scached:      db "Cached Access Time: ",0x00
```

Defines:
`scached`, used in chunk 10d.
`scr`, used in chunks 10e and 11c.

Now we can print the text.

10d ⟨*cachetiming-program* 9c⟩+≡ (12a) ◁10b 10e▷

```
    push      RAX
    mov       RDI,scached
    call      _print
```

Uses `_print` 20a and `scached` 10c.

Then we restore the value and print the measured time to `stdout`. At last we append a CR to the output.

10e ⟨*cachetiming-program* 9c⟩+≡ (12a) ◁10d 11a▷

```
    pop       RDI
    mov       RSI,scratch
    call      _printdu64bit
    mov       RSI,scr
    mov       RDI,1
    call      _nprint
```

Uses `_nprint` 19b, `_printdu64bit` 21a, `scr` 10c, and `scratch` 9a.

Now we do the same with an uncached value. The difference is that we do not load the value before.

11a ⟨*cachetiming-program* 9c⟩+≡ (12a) ◁10e 11c▷

```
mov     RDI,data
clflush [RDI]
lfence
call    _calccachetime
```

Uses `_calccachetime` 8a and `data` 8b.

Now we have the time of the uncached data access in `RAX` and can print it out with some explaining text.

11b ⟨*cachetiming-rodata* 10c⟩+≡ (12a) ◁10c

```
suncached:     db "Uncached Access Time: ",0x00
```

Defines:
`suncached`, used in chunk 11c.

11c ⟨*cachetiming-program* 9c⟩+≡ (12a) ◁11a 11d▷

```
push    RAX
mov     RDI,suncached
call    _print
pop     RDI
mov     RSI,scratch
call    _printdu64bit
mov     RSI,scr
mov     RDI,1
call    _nprint
```

Uses `_nprint` 19b, `_print` 20a, `_printdu64bit` 21a, `scr` 10c, `scratch` 9a, and `suncached` 11b.

At last we exit the program.

11d ⟨*cachetiming-program* 9c⟩+≡ (12a) ◁11c
⟨*exitProgram* 17b⟩

Now we can put everything together and have our `cachetiming` program that we can now execute.

12a ⟨*cachetiming.asm* 12a⟩≡

```
⟨preamble 5⟩

section .rodata
⟨cachetiming-rodata 10c⟩

section .bss
⟨cachetiming-udata 8b⟩

section .text
⟨cachetiming-program 9c⟩

⟨calculate-cache-access-time 8a⟩

⟨xorshift-prng 18a⟩

⟨utilities 17a⟩
```

The program is placed in `asm/`. With `make` in the folder we can create an executable which is moved to `bin/`. There we can execute this program.

```
$ ./cachetiming
Cached Access Time: 72
Uncached Access Time: 372
$
```

2.4 Read Array via Cache Access Time

Now that we have seen that we can determine if a value was in the cache or not (see 2.3 Measure Cache Access Time) we will read a complete array of data by only measuring the cache access time.

For this we start with some data area that we can read later.

12b ⟨*cacheread-preamble* 12b⟩≡ (13b)

```
        datasize        equ 1024
```

12c ⟨*cacheread-udata* 12c⟩≡ (13b) 12d ▷

```
        align           pagesize
        data:           resb datasize
```

Uses `data` 8b and `pagesize` 5.

Additionally we add a scratch area again.

12d ⟨*cacheread-udata* 12c⟩+≡ (13b) ◁ 12c

```
⟨scratch-data 9a⟩
```

TBD

13a ⟨*cacheread-program* 13a⟩≡ (13b)

```
_start:
```

Uses `_start` 5.

TBD

13b ⟨*cacheread.asm* 13b⟩≡

```
⟨preamble 5⟩
⟨cacheread-preamble 12b⟩

section .bss
⟨cacheread-udata 12c⟩

section .text
⟨cacheread-program 13a⟩

⟨exitProgram 17b⟩

⟨calculate-cache-access-time 8a⟩

⟨xorshift-prng 18a⟩

⟨utilities 17a⟩
```

3 Signals

3.1 Basics

TBD

3.2 Detecting Signals

TBD

3.3 Handling Signals

TBD

4 Utilities

4.1 Introduction

TBD

17a ⟨*utilities* 17a⟩≡ (12a 13b)

⟨*nprint* 19b⟩

⟨*print* 20a⟩

⟨*printdu64bit* 21a⟩

4.2 Exit Program

TBD

17b ⟨*exitProgram* 17b⟩≡ (11d 13b)

```
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

18a ⟨*xorshift-prng* 18a⟩≡ (12a 13b) 18b ▷

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

Defines:
_xorshift, used in chunk 9g.

Now we can generate the next 32bit random number.

18b ⟨*xorshift-prng* 18a⟩+≡ (12a 13b) ◁ 18a 19a ▷

```
.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.

19a ⟨*xorshift-prng* 18a⟩+≡ (12a 13b) ◁18b

```
        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

19b ⟨*nprint* 19b⟩≡ (17a)

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

Defines:
_nprint, used in chunks 10e, 11c, 20d, and 23b.

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`

20a ⟨*print* 20a⟩≡ (17a) 20b ▷

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

Defines:
_print, used in chunks 10d and 11c.

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.

20b ⟨*print* 20a⟩+≡ (17a) ◁ 20a 20c ▷

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

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

20c ⟨*print* 20a⟩+≡ (17a) ◁ 20b 20d ▷

```
      sub     RDI,RSI
```

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`.

20d ⟨*print* 20a⟩+≡ (17a) ◁ 20c

```
      call    _nprint
      ret
```

Uses _nprint 19b.

4.5 Printing Numbers

4.5.1 Printing a 64bit Unsigned Integer

The routine `_printdu64bit` print a given number as unsigned decimal number with 64bit to `stdout`.

To print a decimal number we have to divide the number by 10 and get the remainder for printing (from right to left). For this we move the divisor to a register and the dividend to `RAX`. We have to use `RAX` because this is the only register we can use for division.

Additionally we need the address of the scratch area in `RDI` for storing the result. We also save the address of the scratch area to `R8` for later use.

To increment the address during the processing we clear the direction flag.

Parameters

`RDI` the number number to print to `stdout`

`RSI` the address of a scratch area with a size of at least 20 bytes

21a ⟨*printdu64bit* 21a⟩≡ (17a) 21b ▷

```
  _printdu64bit:
        mov       RAX,RDI
        mov       RDI,RSI
        mov       R8,RDI
        mov       RCX,10
        cld
```

Defines:
`_printdu64bit`, used in chunks 10e and 11c.

Now we define a label to jump back when we see that there are still more digits to print. Then we test `RAX` for 0 and end the processing of the digits.

21b ⟨*printdu64bit* 21a⟩+≡ (17a) ◁ 21a 21c ▷

```
  .next:
        cmp       RAX,0
        je        .done
```

Next we divide `RAX` by `RCX`. For this we have to clear `RDX` because this is the higher value of the dividend. The result is then placed into `RAX` and the remainder into `RDX`.

21c ⟨*printdu64bit* 21a⟩+≡ (17a) ◁ 21b 22a ▷

```
        xor       RDX,RDX
        div       RCX
```

We now exchange the result and the remainder because we now need the remainder in RAX (or AL) for further processing. Now we can add the ASCII character '0' to AL and have the correct ASCII value in AL. Now we can store the ASCII character to the scratch area.

22a ⟨*printdu64bit* 21a⟩+≡ (17a) ◁21c 22b▷
```
        xchg    RDX,RAX
        add     AL,'0'
        stosb
```

Now we restore RAX (which we saved to RDX) to go into the next round.

22b ⟨*printdu64bit* 21a⟩+≡ (17a) ◁22a 22c▷
```
        mov     RAX,RDX
        jmp     .next
```

Now that we have all the numbers as ASCII characters we are nearly done. We now have to reverse the number in memory because the number saved at the lowest address is the digit with the least significance.

We now start with checking if we have written any character. If not then we write the ASCII character '0' into the memory. We use the instruction stosb for this to adjust the address in RDI at the same time.

22c ⟨*printdu64bit* 21a⟩+≡ (17a) ◁22b 22d▷
```
  .done:
        cmp     RDI,RSI
        jne     .printout
        mov     AL,'0'
        stosb
  .printout:
```

Next we calculate the number of digits that the number has. For this we move the address of the last digit to RDX and subtract the start of the scratch area from this. Next we adjust RDI because it points to the first address after the number.

22d ⟨*printdu64bit* 21a⟩+≡ (17a) ◁22c 23a▷
```
        mov     RDX,RDI
        sub     RDX,RSI
        dec     RDI
```

We now have RSI with the address of the start of the number and RDI with the address of the end. We now have to exchange the digits from the front and the end to get the right number. For this we increment RSI and decrement RDI after each exchange and when the addresses pass each other we are done.

23a ⟨*printdu64bit* 21a⟩+≡ (17a) ◁22d 23b▷

```
.reverse:
     mov       AL,[RSI]
     mov       AH,[RDI]
     mov       [RSI],AH
     mov       [RDI],AL
     dec       RDI
     inc       RSI
     cmp       RSI,RDI
     jb        .reverse
```

Now we restore the address of the scratch area to RSI and move the number of digits (which we stored in RDX) to RDI and can the call _nprint to print the number.

23b ⟨*printdu64bit* 21a⟩+≡ (17a) ◁23a

```
     mov       RSI,R8
     mov       RDI,RDX
     call      _nprint
     ret
```

Uses _nprint 19b.

A Glossary

x86 x86 denotes a microprocessor architecture based on the 8086/8088 18

B Acronyms

ASCII American Standard Code for Information Interchange 22

CR carriage return 10

RNG random number generator 18

C x86-Instructions

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

`lfence` Load Fence, introduced with Intel® Pentium® 4 8, 10

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

D Code Chunks