CISPA
HELMHOLTZ CENTER FOR
INFORMATION SECURITY

Timing Instructions for Fun and Profit

Daniel Weber, Lukas Gerlach, Michael Schwarz | August 23, 2022

Agenda

High-Precision Timings

Distinguish Instructions

Break Security Mitigations

Exercise Requirements

Hardware/Software Requirements:

- x86 CPU (Intel or AMD)
- Linux installation
- Installed tools: `python3`, `gcc`, `make`
- Installed Python package: `matplotlib`

Translating Code to Assembly

C Code

```
unsigned int a = 5;
unsigned int b = 2;

return a * b;
```

Option 1

```
mov eax, 5
mov ebx, 2

mul ebx
ret
```

Option 2

```
mov eax, 5
mov ebx, 2

add eax, eax
ret
```

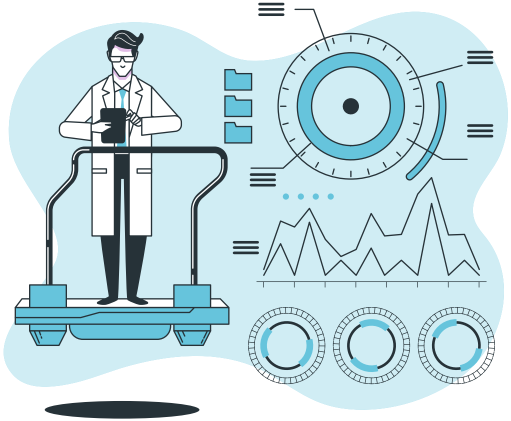
Different Execution Times?

Yes!

Different Execution Times

Instructions vary in their execution time

• `NOP`	$\Longrightarrow$	**instant** (“0” CPU cycles)
• `ADD` / `XOR`	$\Longrightarrow$	**fast** (1 CPU cycle)
• `MUL`	$\Longrightarrow$	**okish** (3-4 CPU cycles)
• `SYSCALL` / `DIV`	$\Longrightarrow$	**slow** (10-39 CPU cycles)

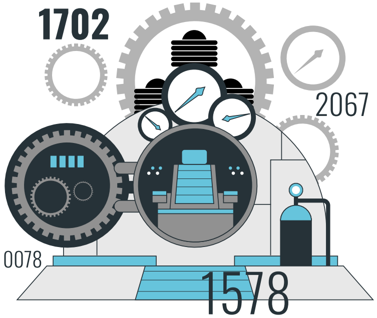


Can we **observe** these effects?

Let’s find out!

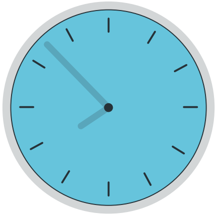
Experiment Idea

1) Time fast instruction
2) Time slow instruction
3) Plot the timings



How do we get **high-precision time measurements**?

High-Precision Time Measurements



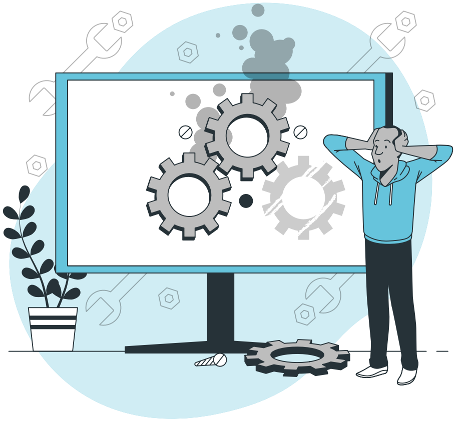
- x86 has two **instructions**: `rdtsc` and `rdtscp`
- Reads the processor's **time-stamp counter**

→ CPU cycles since reset

- Highly accurate (**nanoseconds**), low overhead

High-Precision Time Measurements

```
[...]
rdtsc
function()
rdtsc
[...]
```



What about out-of-order execution?

Timings can be Reordered

Out-of-order execution $\rightarrow$ different possibilities

```
rdtsc                   rdtsc                   rdtsc
function()              [...]                   rdtsc
[...]                   rdtsc                   function()
rdtsc                   function()              [...]
```

Prevent Reordering

- **Pseudo-serializing** instruction `rdtscp` (recent CPUs)
- **Serializing** instructions like `cpuid`
- **Fences** like `mfence`

Intel, *How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures White Paper*, December 2010.

High-Precision Time Measurements (Accurate)

```
[...]
mfence
rdtsc
mfence
function()
mfence
rdtsc
mfence
[...]
```

Let’s get our hands dirty!

Exercise 1: Instruction Timing Differences

The Task:

Build a histogram showing the difference between a `XOR` and a `DIV`.

Hints for better results:

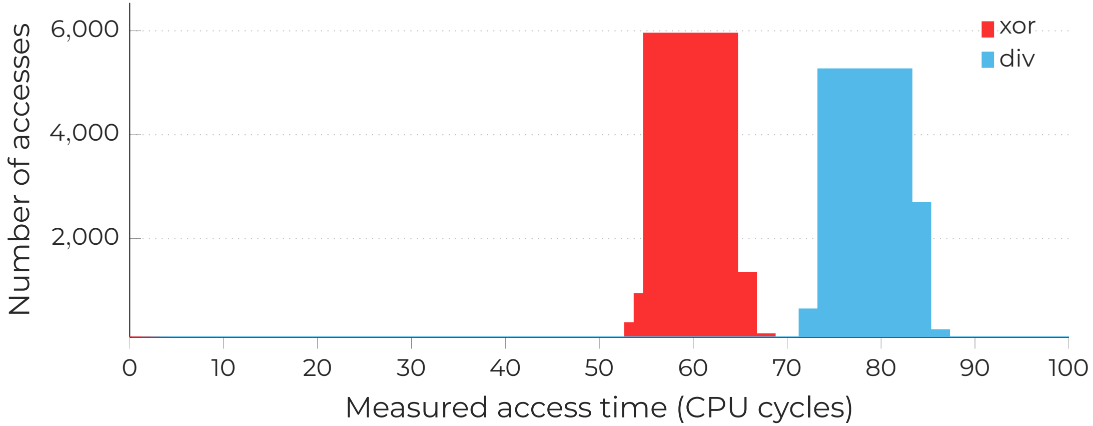
- Connect your laptop to power
- Close unrelated programs

Exercise 1:

Observing Instruction Timings

`https://challenge.attacking.systems/timing.tar.gz`

Result: Instruction Timings

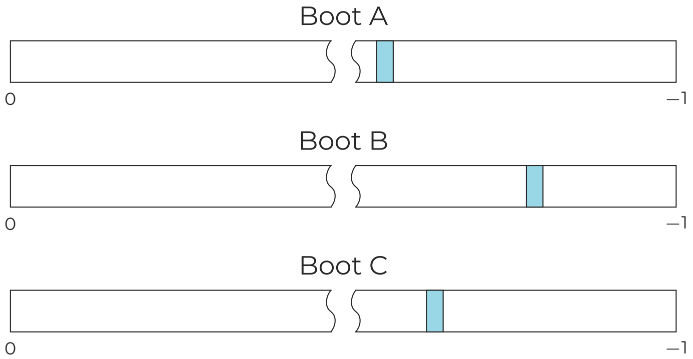


Can we do something with that?

Kernel ASLR (KASLR)

- Many exploits rely on the **knowledge of the memory location** of a certain function
- OS Kernels are **protected by KASLR**
- Memory location of the kernel is **randomized**
- Attacker **do not know where** to attack

KASLR: Kernel Address Space Layout Randomization



Kernel is loaded to a **different** offset on every boot

Prefetch

PREFETCH (Intel Optimization Ref. Manual – Chapter 9)

Using the PREFETCH instruction is recommended only if data does not fit in cache. Use of software prefetch should be limited to memory addresses that are managed or owned within the application context. Prefetching to addresses that are **not mapped** to physical pages can experience **non-deterministic** performance penalty. For example, specifying a NULL pointer (0L) as address for a prefetch can cause **long delays**.

The PREFETCH instruction executes **faster** when **accessing a mapped kernel address**.

Exercise 2: Breaking KASLR with Prefetch

The Task:

Iterate over the possible kernel locations.

For each location:

→ Access the address with `prefetch`

→ Get the access timing

→ Print the address of the first fast access

Hints for better results:

- Connect your laptop to power
- Noise can be helpful: `stress -c 1 -m 1 &`

Exercise 2:

Breaking KASLR with Prefetch

`https:///challenge.attacking.systems/KASLR.tar.gz`