Module 7: Cloud Security
side-channel attacks and countermeasures

Meng Xu (University of Waterloo)
Spring 2023
1. What is a side-channel?

2. Timing-based cache side channels

3. Covert channels
How to steal sensitive information?

- Install a malware (spyware) on the victim’s computing device
  - e.g., screen hijacking, drive-by downloads

- Exploit a vulnerability in victim’s software
  - e.g., heartbleed, log4j, etc.

- Compromised operating system, hypervisor, or hardware
  - e.g., key logger, buggy virtualization layer, etc.

- Side channels
  - e.g., timing, bandwidth, power, etc.
Locard’s exchange principle

In forensic science, **Locard’s principle** holds that: the perpetrator of a crime will bring something into the crime scene and leave with something from it, and that both can be used as forensic evidence.

→ “Every contact leaves a trace”

*Wherever he steps, whatever he touches, whatever he leaves, even unconsciously, will serve as a silent witness against him. Not only his fingerprints or his footprints, but his hair, the fibres from his clothes, the glass he breaks, the tool mark he leaves, the paint he scratches, the blood or semen he deposits or collects. All of these and more, bear mute witness against him. This is evidence that does not forget.*

— Paul L. Kirk
Locard’s exchange principle (in code execution)

In forensic science, **Locard’s principle** holds that: the perpetrator of a crime execution of code will bring something into the crime scene hosting platform and leave with something from it, and that both can be used as forensic evidence side channels.

→ “Every contact leaves a trace”

Wherever he steps **Every CPU instruction executed**, **whatever he touches every memory access**, **whatever he leaves every IO operation**, even unconsciously, will serve as a silent witness against him the code.
My personal story
Examples of side channels

- Bandwidth consumption
- Reflections
- Cache-timing channels
Bandwidth consumption: scenario

- Eve observes communication going via Alice’s Router
- Alice accesses health forum via encrypted connection
- Eve knows that Alice connects to health forum
- But cannot decrypt downloaded content
Eve determines size of all pages on health forum
Eve measures size of Alice’s downloaded pages
Likely: Eve can uniquely map download to page
This attack is called *webpage fingerprinting*
  - or *website fingerprinting*, when targeting landing pages
Bandwidth consumption: defense

- Pad all pages to common size (inflexible + inefficient 😞)
- Dynamic personalized websites
- (Finally a benefit of targeted advertisement)
Bandwidth consumption: another example

- Re-identification of Netflix video streaming
- Burst sizes of a streamed scene of “Reservoir Dogs”
  - Very similar, even when watched over different networks

Schuster et al., USENIX SEC '17
Reflections: scenario

- Alice types her password on a device in a public place
- Alice hides her screen
- But there is a reflecting surface close by
Reflections: attack

- Eve uses a camera and a telescope
- Off-the-shelf: less than CA$2,000
- Photograph reflection of screen through telescope
- Reconstruct original image
- Distance: 10–30 m
- Depends on equipment and type of reflecting surface
Reflections: defense
Other potential attack vectors

- Timing computations
- Electromagnetic emission
- Sound emissions
- Power consumption
- Differential power analysis
- Differential fault analysis
Outline

1. What is a side-channel?
2. Timing-based cache side channels
3. Covert channels
Common hardware shared

- Memory
  - Memory deduplication
  - Memory bus
  - DRAM row buffer

- CPU
  - Branch prediction
  - Data cache
  - Instruction cache

Shared Hardware
Cache timing side channels

- Modern architectures use caches to speed up memory access
  - Main memory access is slow. Cache access is faster.
  - Caches are micro-architectural objects, not architectural: programs typically unaware of caches.
  - Caches are shared: by timing cache access, a process can learn information about data used by another.

- Micro-architectural features like speculative and out-of-order execution can be exploited to leak information via caches.
  - Spectre and Meltdown attacks (2017)
Why targeting cache?

- Shared across cores
  - agonistic to kernel, hypervisor, and emulators!

- Observable effect
  - data/instruction is cached → cache hit → fast
  - data/instruction is not cached → cache miss → slow

⇒ cross-core attacks!
Measuring time differences caused by cache

1. build two cases: cache hits and cache misses
2. time each case many times (get rid of noise)
3. plot a histogram
4. find a threshold to distinguish the two cases
Side note: how to measure timing accurately?

rdtsc instruction: cycle-accurate timestamps

t1 := rdtsc
⟨...operation...⟩
t2 := rdtsc
duration := t2 - t1

Q: Is this correct?
Do you measure what you think you measure?
Side note: how to measure timing accurately?

A: Out-of-order execution

\[
\begin{align*}
\text{t1} & := \text{rdtsc} \\
\langle \ldots \text{operation} \ldots \rangle & \\
\text{t2} & := \text{rdtsc} \\
\langle \ldots \text{other-ops} \ldots \rangle & \\
\text{duration} & := \text{t2} - \text{t1} \\
\langle \ldots \text{other-ops} \ldots \rangle & \\
\text{t1} & := \text{rdtsc} \\
\langle \ldots \text{other-ops} \ldots \rangle & \\
\text{t2} & := \text{rdtsc} \\
\langle \ldots \text{operation} \ldots \rangle & \\
\text{duration} & := \text{t2} - \text{t1} \\
\langle \ldots \text{operation} \ldots \rangle & \\
\text{t1} & := \text{rdtsc} \\
\langle \ldots \text{operation} \ldots \rangle & \\
\text{t2} & := \text{rdtsc} \\
\langle \ldots \text{other-ops} \ldots \rangle & \\
\text{duration} & := \text{t2} - \text{t1} \\
\langle \ldots \text{other-ops} \ldots \rangle & 
\end{align*}
\]
Side note: how to measure timing accurately?

- use pseudo-serializing instruction `rdtscp` (recent CPUs)
- and/or use serializing instructions like `cpuid`
- and/or use fences like `mfence`

Timing difference on cache hit/miss

![Histogram of cache hits and misses](image.png)
Cycle difference on cache hit/miss

On current Intel CPUs:

- L1 cache: 4 cycles
- L2 cache: 12 cycles
- L3 cache: 26-31 cycles
- DRAM memory: > 120 cycles
(Unprivileged) cache maintenance

User programs can (voluntarily) optimize cache usage:
- `prefetch`: suggest CPU to load data into cache
- `clflush`: throw out data from all caches
...based on virtual addresses

This is the enabler of any cache-based attack:
Attacker monitors its own activity to find cache accessed by victim.
A concrete scenario

- You run a secret program on machine, and the program does one of two things:
  - encrypt()
  - decrypt()
- You do not want anyone to know whether your program is encrypting a message or decrypting a message:
  - assuming trust in operating system and hardware
- The binary of your program is available.

- Attackers can run their programs on the same machine.
- Their goal is to infer which operation your program is running.
Access-driven attacks

- **Flush + Reload**

- **Prime + Probe**
Flush+Reload

Init: victim program loaded while cache is empty  
Step 1: attacker loads the encrypt() code into its address space  
Step 2: attacker flushes the shared cache

Step 3(a): victim performs operation encrypt()  
Step 3(b): victim performs operation decrypt()  
Step 4: attacker calls encrypt() after step 3(a)  

Step 4: attacker calls encrypt() after step 3(b)  

**Prime+Probe**

**Init:** victim program loaded while cache is empty

**Step 1:** attacker fills all available cache (prime)

Step 2(a): victim evicts cache lines while performing operation encrypt()

Step 2(b): victim evicts cache lines while performing operation decrypt()

**Step 3:** attacker calls encrypt() after step 2(a) = fast!

**Step 3:** attacker calls encrypt() after step 2(b) = slow!
Potential countermeasures

- Fuse the functionalities into a single code piece

```plaintext
1 if (flag) { encrypt_instruction_1(); }
2 else { decrypt_instruction_1(); }
3 if (flag) { encrypt_instruction_2(); }
4 else { decrypt_instruction_2(); }
5 ...
6 if (flag) { encrypt_instruction_N(); }
7 else { decrypt_instruction_N(); }
```

- Voluntary cache eviction

Both at the expense of performance overhead.
More systematic countermeasures

- Use constant-time programming techniques
- Eliminate secret-dependent execution or branches
- Hide/blind inputs
More on constant time techniques

```c
void foo(double x) {
    double z, y = 1.0;
    for (long i = 0; i < 100000000; i++) {
        z = y * x;
    }
}
```

**Q**: Which of the following execution is faster?

1. `foo(1.0)`
2. `foo(1.0e-323)`
3. they are the same

**A**: `foo(1.0)`

Some observations:
- Avoid variable-time instructions
- If-statements on secrets are unsafe
- There are tools to help but most constant-time code is still
Outline

1. What is a side-channel?
2. Timing-based cache side channels
3. Covert channels
How to secretly exchange information?

An attacker creates a capability to transfer sensitive/unauthorized information through a channel that is not supposed to transmit that information.

What information can/cannot be transmitted through a channel may be determined by a policy/guidelines/physical limitations, etc.
How can we create a covert channel?

- Assume that Eve can arrange for malicious code to be running on Alice’s machine
  - But Alice closely watches all Internet traffic from her computer
  - Better, she doesn’t connect her computer to the Internet at all!

- Suppose Alice publishes a weekly report summarizing some (nonsensitive) statistics

- Eve can “hide” the sensitive data in that report!
  - e.g., modifications to spacing, wording, or the statistics itself
End