Cache Attacks, continued!

Lab tonight! Building the basis of a cache attack; Potential additional colloquium opportunity, Tuesday, October 28 at Mudd ➡️ “(Re)computing Electronic Waste”

FEBRUARY 26, 2024

Statements of Support for Software Measurability and Memory Safety

ONCD ▸ BRIEFING ROOM ▸ PRESS RELEASE

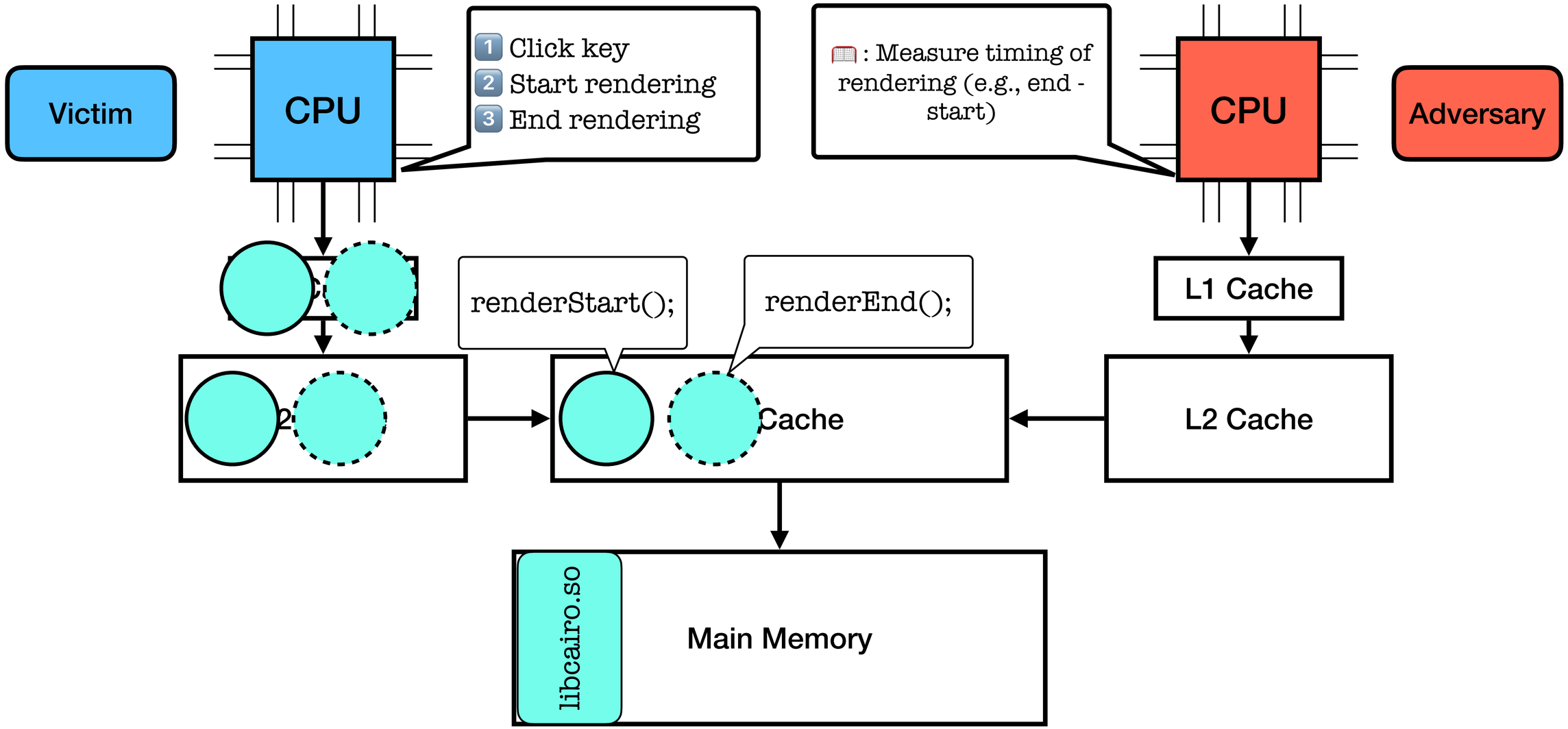
ABSTRACT

President Biden's National Cybersecurity Strategy outlines two fundamental shifts: the need to both rebalance the responsibility to defend cyberspace and realign incentives to favor long-term cybersecurity investments. In this report, the case is made that the technical community is well-positioned to drive progress on both strategic goals. First, in order to reduce memory safety vulnerabilities at scale, creators of software and hardware can better secure the building blocks of cyberspace. This report focuses on the programming language as a primary building block, and explores hardware architecture and formal methods as complementary approaches to achieve similar outcomes. Second, in order to establish accurate cybersecurity quality metrics, advances can be made to address the hard and complex research problem of software measurability. This report explores how such metrics can shift market forces to improve cybersecurity quality across the ecosystem.

Outline

- Revisiting the attack setup
- Introducing timing side channel attack methodologies
- Building protected caches

Shared Memory Model



Measuring Shared Cache Behavior

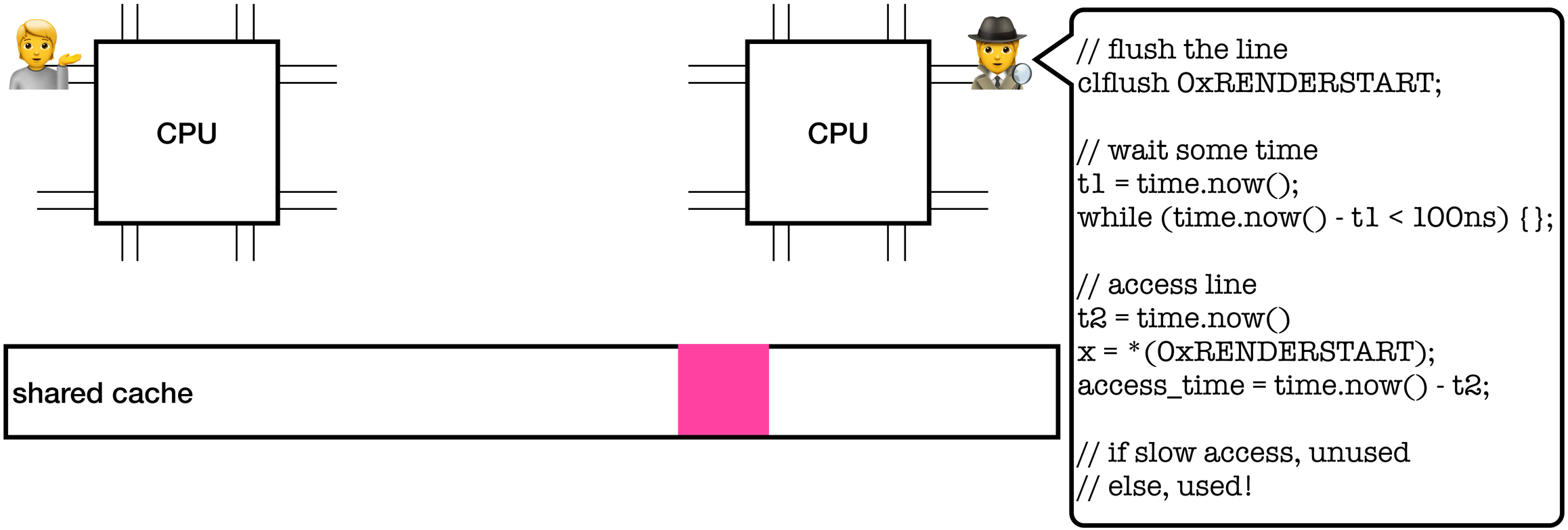
- We want to be able to say when certain behaviors have occurred in the victim process based on events that we can observe in shared hardware
- If we want to know that a victim has called a function, then we would expect that address of that function will be in the shared caches (as it has been loaded into the L1 instruction cache the victim processor)
- To find this value, the adversary should also try accessing this address to see if it exists in the shared cache

Chat with your neighbor(s)!

Describe a case where the victim calls `renderStart` and the data is not observable by the adversary. What leads to this case? Are there other cache features that the adversary can use to achieve the same end?

The adversary needs to be able to control what is in the victim's cache state to be able to infer what they accessed!

Flush + Reload Attack

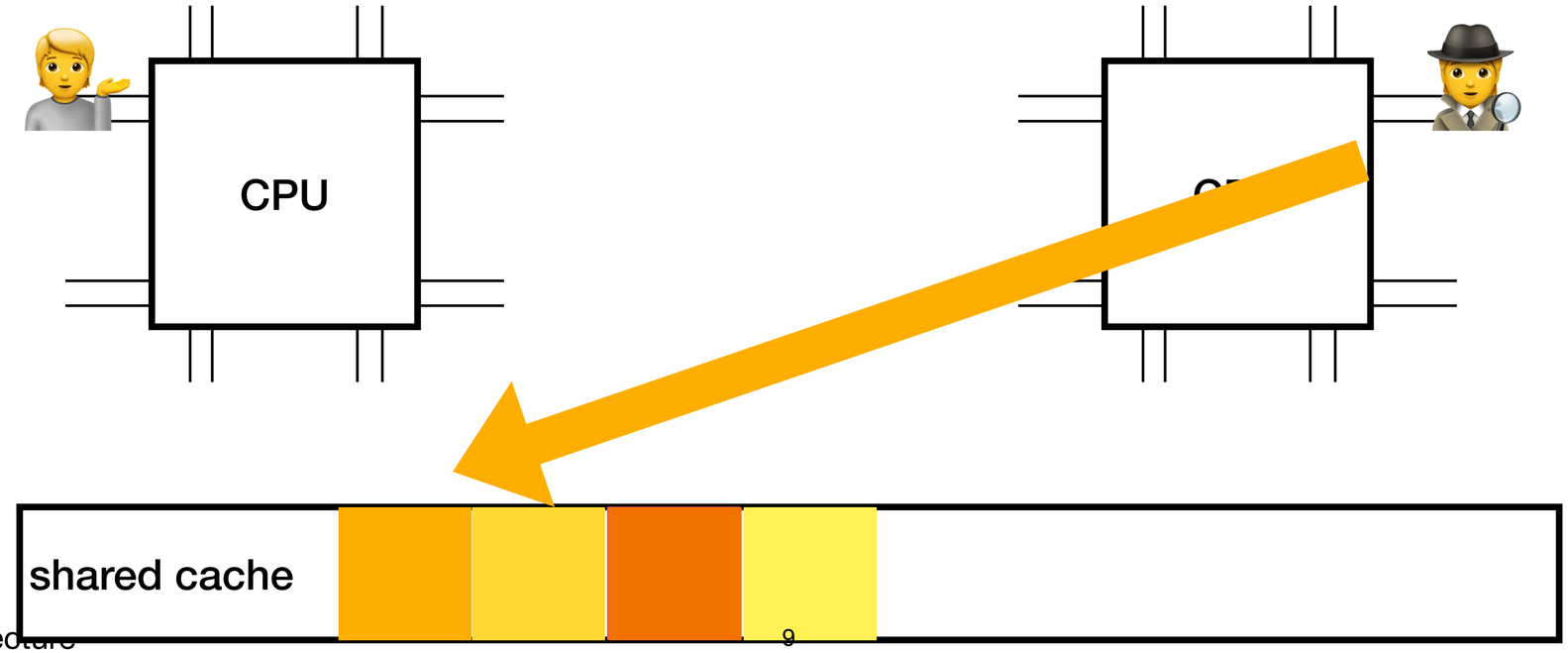


Flush + Reload Attack

- For some sensitive data, evict it from the cache state ➡️ this means that any subsequent hit on that data was a due to the victim's behavior
- To implement such an attack, an adversary needs to be able to use very precise timers (nanosecond granularity) and be able to execute completely in parallel with the victim
- x86 gives access to the `clflush` instruction to be able to explicitly interact with long term storage and write certain data through to non-volatile storage
- Potential mitigation: What happens if the ISA and/or hardware doesn't allow for explicit flushing instructions or this granularity of timing?

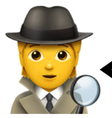
Prime + Probe Attack

- Unfortunately, we are not safe with these ISA defenses!
- While, we may not be able to time individual accesses, but we can measure the impact of misses due to *cache contention*



Prime + Probe Attack

- Performing a prime + probe achieves the same end of evicting the data, but does not require any special instructions!
- An adversary can control what is in the cache state by filling it with data that occupies all available blocks ➡️ how might they be able to achieve this? (You will do this in lab tonight!)
- The granularity of timing becomes significantly less precise ➡️ priming the cache is slow by definition!



```
// flush the line
clflush 0xRENDERSTART;

// wait some time
t1 = time.now();
while (time.now() - t1 < 100ns) {};

// access line
t2 = time.now()
x = *(0xRENDERSTART);
access_time = time.now() - t2;

// if slow access, unused
// else, used!
```

Cache Timing Side Channel Takeaways

- Data shared by multiple applications (e.g., shared library functions) are visible throughout the cache hierarchy and memory system by all sharing processes
- To achieve the intended illusion of a “large, fast memory”, the cache hierarchy is constructed to allow for recently accessed data to be accessed at a lower latency!
- If there are observable behaviors (even if unintentional) in the memory system that leak information, this can be exploited to infer sensitive behaviors being performed in the victim application

So, how effective was the attack?

TABLE III: Example dictionary-assisted password guessing attack for password “hello”.

Input	Confidence Vector (Partial)							
	e	h	i	j	l	o	s	y
h	0.0	0.39	0.0	0.23	0.0	0.0	0.0	0.03
e	0.21	0.0	0.0	0.0	0.0	0.0	0.0	0.03
l	0.0	0.0	0.05	0.0	0.37	0.0	0.07	0.0
l	0.0	0.0	0.05	0.0	0.37	0.0	0.07	0.06
o	0.0	0.0	0.0	0.0	0.0	0.15	0.0	0.0

Rank	Dictionary Words	Confidence Value
1	hello	$0.39 + 0.21 + 0.37 + 0.37 + 0.15 = 1.49$
2	jelly	$0.23 + 0.21 + 0.37 + 0.37 + 0.0 = 1.18$
3	hills	$0.39 + 0.0 + 0.37 + 0.37 + 0.0 = 1.13$
4	holly	$0.39 + 0.0 + 0.37 + 0.37 + 0.0 = 1.13$

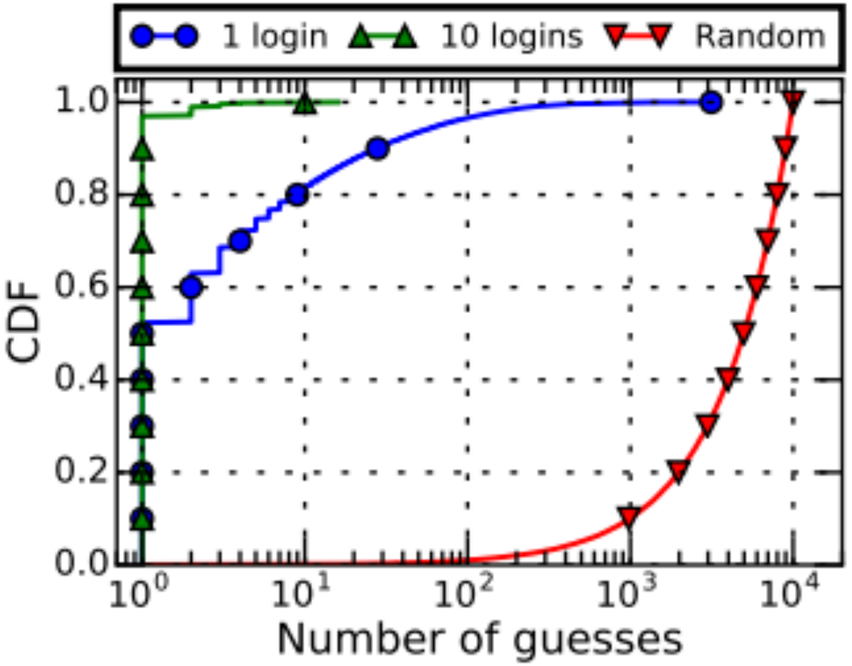


Image credit: https://www.ndss-symposium.org/ndss-paper/unveiling-your-keystrokes-a-cache-based-side-channel-attack-on-graphics-libraries/

Chat with your neighbor(s)!

Having observed the features that go into developing this attack, think about how you would go about building a defense. What would be the features that you would target first?

Defending Against Shared Cache Attacks

- Keeping instructions private: if the `renderStart` and `renderEnd` instructions are at private addresses rather than shared addresses, then the adversary will not necessarily know which addresses to look for
- Keeping cache state private (or partitioned): if caches have certain regions delegated to certain processes/processors, then the behavior of the victim and adversary are not intertwined
- Mitigating accesses with noisy/random accesses: if the cache makes some accesses that do not correspond to application behavior, then an adversary would not be able to infer whether a hit corresponds to the victim actually calling `renderStart`
- Hiding application leakage: if the timing to render different characters is consistent for all characters, then the differences between start and end time should not leak the keystroke information!