Lecture #7: ASLR

UCalgary ENSF619

Elements of Software Security

Instructor: Lorenzo De Carli (lorenzo.decarli@ucalgary.ca)

Slides partly based on SysBumps, Jang et al., ACM CCS 2024

Today’s lecture is about KASLR

- But **what is it**?
- First, let’s refresh our knowledge of what **ASLR** is in the first place

- Observation: most memory exploits work by **causing execution to jump into a memory region which contains useful code**
  - Typically **libc** or the **executable itself**
  - Jumping elsewhere risks ending up into **unmapped** or **non-X** memory

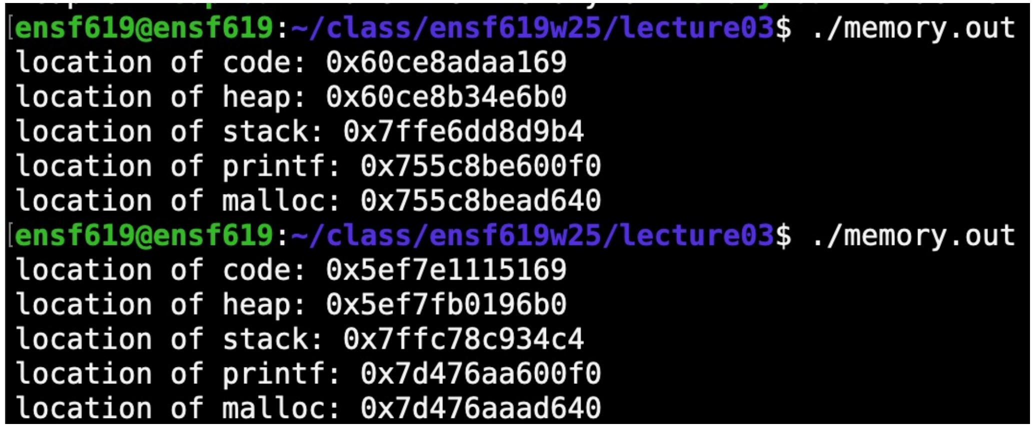
What does ASLR do?

- The previous observation suggests an **insight**:
  - If we map relevant **memory regions** at **random offsets**…
  - …then exploit writers **won't be able to jump** at the correct location

- ASLR **randomizes the location** of **relevant memory regions**
  - Typically **stack**, **heap**, **text**, **libraries**
- Attacker must get creative to guess the right location **before an exploit can be carried**

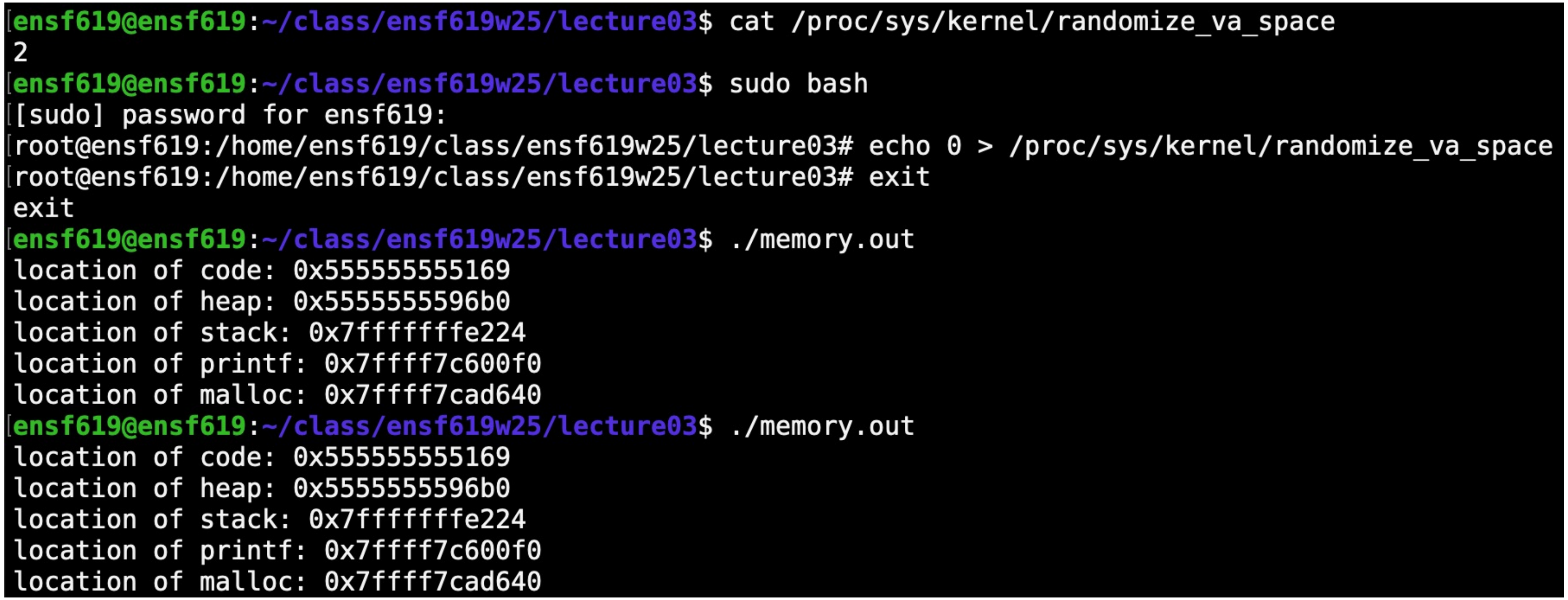
ASLR - example

With ASLR



```
[ensf619@ensf619:~/class/ensf619w25/lecture03$ ./memory.out
location of code: 0x60ce8adaa169
location of heap: 0x60ce8b34e6b0
location of stack: 0x7ffe6dd8d9b4
location of printf: 0x755c8be600f0
location of malloc: 0x755c8bead640
[ensf619@ensf619:~/class/ensf619w25/lecture03$ ./memory.out
location of code: 0x5ef7e1115169
location of heap: 0x5ef7fb0196b0
location of stack: 0x7ffc78c934c4
location of printf: 0x7d476aa600f0
location of malloc: 0x7d476aaad640
```

Without ASLR



```
[ensf619@ensf619:~/class/ensf619w25/lecture03$ cat /proc/sys/kernel/randomize_va_space
2
[ensf619@ensf619:~/class/ensf619w25/lecture03$ sudo bash
[[sudo] password for ensf619:
[root@ensf619:/home/ensf619/class/ensf619w25/lecture03# echo 0 > /proc/sys/kernel/randomize_va_space
[root@ensf619:/home/ensf619/class/ensf619w25/lecture03# exit
exit
[ensf619@ensf619:~/class/ensf619w25/lecture03$ ./memory.out
location of code: 0x555555555169
location of heap: 0x5555555596b0
location of stack: 0x7fffffffe224
location of printf: 0x7ffff7c600f0
location of malloc: 0x7ffff7cad640
[ensf619@ensf619:~/class/ensf619w25/lecture03$ ./memory.out
location of code: 0x555555555169
location of heap: 0x5555555596b0
location of stack: 0x7fffffffe224
location of printf: 0x7ffff7c600f0
location of malloc: 0x7ffff7cad640
```

What is KASLR?

- Similar idea, but **randomize the OS Kernel memory** region
- Why do we need to worry about this?
- The kernel cannot be exploited, right? **RIGHT?**

- …turns out, **memory exploits are possible in kernel space too!**

How do kernel exploits work?

- They can work in **many different ways**, but…
- …typically the idea is some **vulnerable kernel function** is identified, that **receives data from userspace**
- By passing malformed data, it is possible to accomplish:
  - **Stack overflows**
  - **Heap overflows**
  - **Arbitrary memory writes**
- These attacks can in turn be used for example to **raise privileges**

How does ASLR look when applied to kernel?

- **Kernel memory cannot be as easily randomized** as a user programs
  - Hardware specifications ”block” certain addresses that **cannot thus be moved easily**
- ”Randomized” ends up being **milder** than in the userspace case
- In practice, the **base address of the kernel** is **randomized**, but the rest **stays constant**

How does it look in the case of MacOSX? (from today's paper)

$$Kernel\_base = \texttt{0xfffffe0007004000} + slide.$$

Base address of kernel

Constant offset

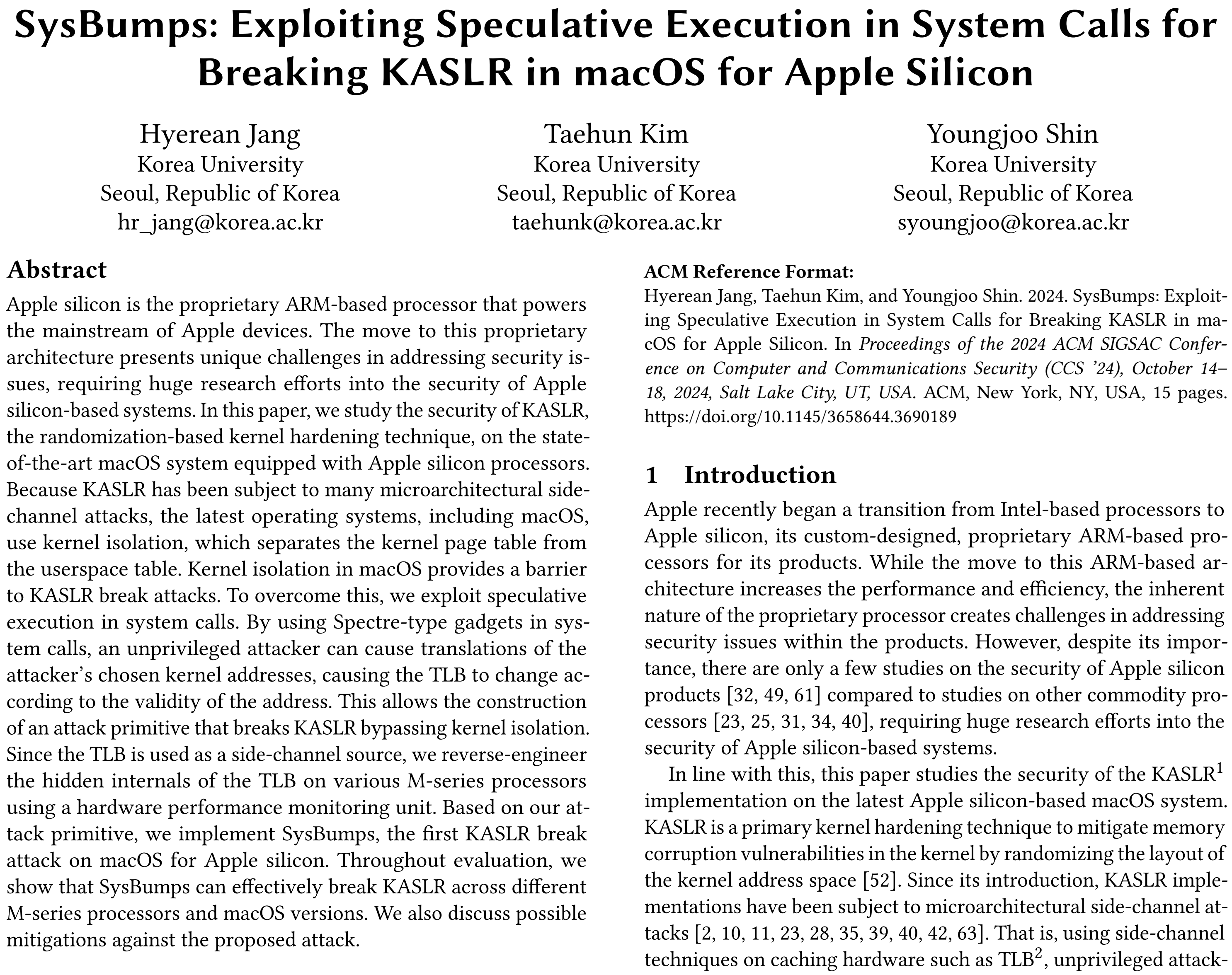
Randomized value

- **Overall offset** aligned to **16KB** (system page size)
- **Highest and lowest base address** determined through **repeated measures**
- **Example**(M2 Max processor):
  $0xFFFFFE002F000000 - 0xFFFFFE000F1C4000 = 0x1FE3C000 = 535019520$
  $535019520 / 16384 = 32655 \approx 2^{15} \rightarrow$ *15 bits of randomness*

In a nutshell…

- **Defeating KASLR** entails **determining the kernel base address**
- If that address is discovered, **KASLR is “broken”**
- If I have a kernel-level exploit that requires knowledge of the kernel memory location, **I can now carry it**

Let’s talk about the paper



SysBumps: Exploiting Speculative Execution in System Calls for Breaking KASLR in macOS for Apple Silicon

Hyerean Jang
Korea University
Seoul, Republic of Korea
hr_jang@korea.ac.kr

Taehun Kim
Korea University
Seoul, Republic of Korea
taehunk@korea.ac.kr

Youngjoo Shin
Korea University
Seoul, Republic of Korea
syoungjoo@korea.ac.kr

Abstract

Apple silicon is the proprietary ARM-based processor that powers the mainstream of Apple devices. The move to this proprietary architecture presents unique challenges in addressing security issues, requiring huge research efforts into the security of Apple silicon-based systems. In this paper, we study the security of KASLR, the randomization-based kernel hardening technique, on the state-of-the-art macOS system equipped with Apple silicon processors. Because KASLR has been subject to many microarchitectural side-channel attacks, the latest operating systems, including macOS, use kernel isolation, which separates the kernel page table from the userspace table. Kernel isolation in macOS provides a barrier to KASLR break attacks. To overcome this, we exploit speculative execution in system calls. By using Spectre-type gadgets in system calls, an unprivileged attacker can cause translations of the attacker’s chosen kernel addresses, causing the TLB to change according to the validity of the address. This allows the construction of an attack primitive that breaks KASLR bypassing kernel isolation. Since the TLB is used as a side-channel source, we reverse-engineer the hidden internals of the TLB on various M-series processors using a hardware performance monitoring unit. Based on our attack primitive, we implement SysBumps, the first KASLR break attack on macOS for Apple silicon. Throughout evaluation, we show that SysBumps can effectively break KASLR across different M-series processors and macOS versions. We also discuss possible mitigations against the proposed attack.

ACM Reference Format:
Hyerean Jang, Taehun Kim, and Youngjoo Shin. 2024. SysBumps: Exploiting Speculative Execution in System Calls for Breaking KASLR in macOS for Apple Silicon. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS ’24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690189

1 Introduction

Apple recently began a transition from Intel-based processors to Apple silicon, its custom-designed, proprietary ARM-based processors for its products. While the move to this ARM-based architecture increases the performance and efficiency, the inherent nature of the proprietary processor creates challenges in addressing security issues within the products. However, despite its importance, there are only a few studies on the security of Apple silicon products [32, 49, 61] compared to studies on other commodity processors [23, 25, 31, 34, 40], requiring huge research efforts into the security of Apple silicon-based systems.

In line with this, this paper studies the security of the KASLR[1] implementation on the latest Apple silicon-based macOS system. KASLR is a primary kernel hardening technique to mitigate memory corruption vulnerabilities in the kernel by randomizing the layout of the kernel address space [52]. Since its introduction, KASLR implementations have been subject to microarchitectural side-channel attacks [2, 10, 11, 23, 28, 35, 39, 40, 42, 63]. That is, using side-channel techniques on caching hardware such as TLB[2], unprivileged attack-

Workplan for this paper

1. Find a way to cause kernel to access memory address
2. Measure which addresses are valid
3. Find location of kernel in memory

Why do we care about TLB?

- We need to find a way to determine whether **any address within the kernel address space is valid (mapped) or not**
- **Impossible** to do this directly from **user space**
  - User space applications **cannot access kernel memory**!
- Must use an **indirect approach**

The way in

- Certain system calls receive **pointers** as **parameters**
- **General idea: pass memory addresses** to those calls and **determine if they are valid or not** by looking at how the kernel behaves
- **Problems:**
  1. The kernel won't even try to access those addresses, as it will immediately realize they are invalid
  2. Even if the kernel does try to access those addresses, how do I observe its behavior?
  3. Finally, even if I can observe kernel behavior, how do I use this to break ASLR?

Problem 1: get kernel to access invalid addresses

- MacOS system calls are hardened against incorrect input

```
int copyinstr(const user_addr_t user_addr, char *kernel_addr, vm_size_t nbytes, vm_size_t *lencopied)
{
    int result;
    ...
    result = copy_validate(user_addr, (uintptr_t)kernel_addr, nbytes, COPYIO_IN);
    if (__improbable(result)) {
        // When user_addr is invalid
        return result;
    }

    // When user_addr is valid
    user_access_enable();
    result = _bcopyinstr((const char *)user_addr, kernel_addr, nbytes, &bytes_copied);
    user_access_disable();
    ...
}
```

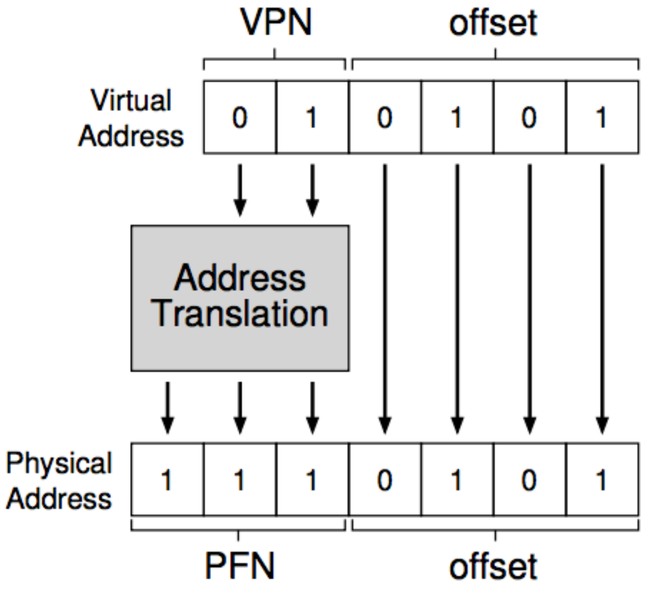
Passing an arbitrary kernel address in place of user_addr will cause this check to fail!

Solution 1: take advantage of speculative execution

- Modern CPUs are **very efficient**
- To save time, they will run **branch prediction** and **speculatively execute** instructions on the most likely side of the branch
- If it turns out the **prediction is incorrect**, the **effect** of those instructions will be **rolled back**
- **…or, will it?**

A speedy intro to the TLB

- With **virtual memory**, each time a memory access is performed, the **virtual address** must be translated to **a physical address**
- **Approach**: Cache recent translations in the **translation look-aside buffer (TLB)** to avoid costly accesses to the page table.



The issue with caching

- Turns out, certain changes to the content of the TLB, caused by mispredicted instructions, will persist even when the instruction is rolled back

```
    if (__improbable(result)) {
        // When user_addr is invalid
        return result;
    }

    // When user_addr is valid
    user_access_enable();
    result = _bcopyinstr((const char *)user_addr, kernel_addr, nbytes, &bytes_copied);
    user_access_disable();
    ...
}
```

If the branch predictor thinks this if () is going to evaluate to FALSE…

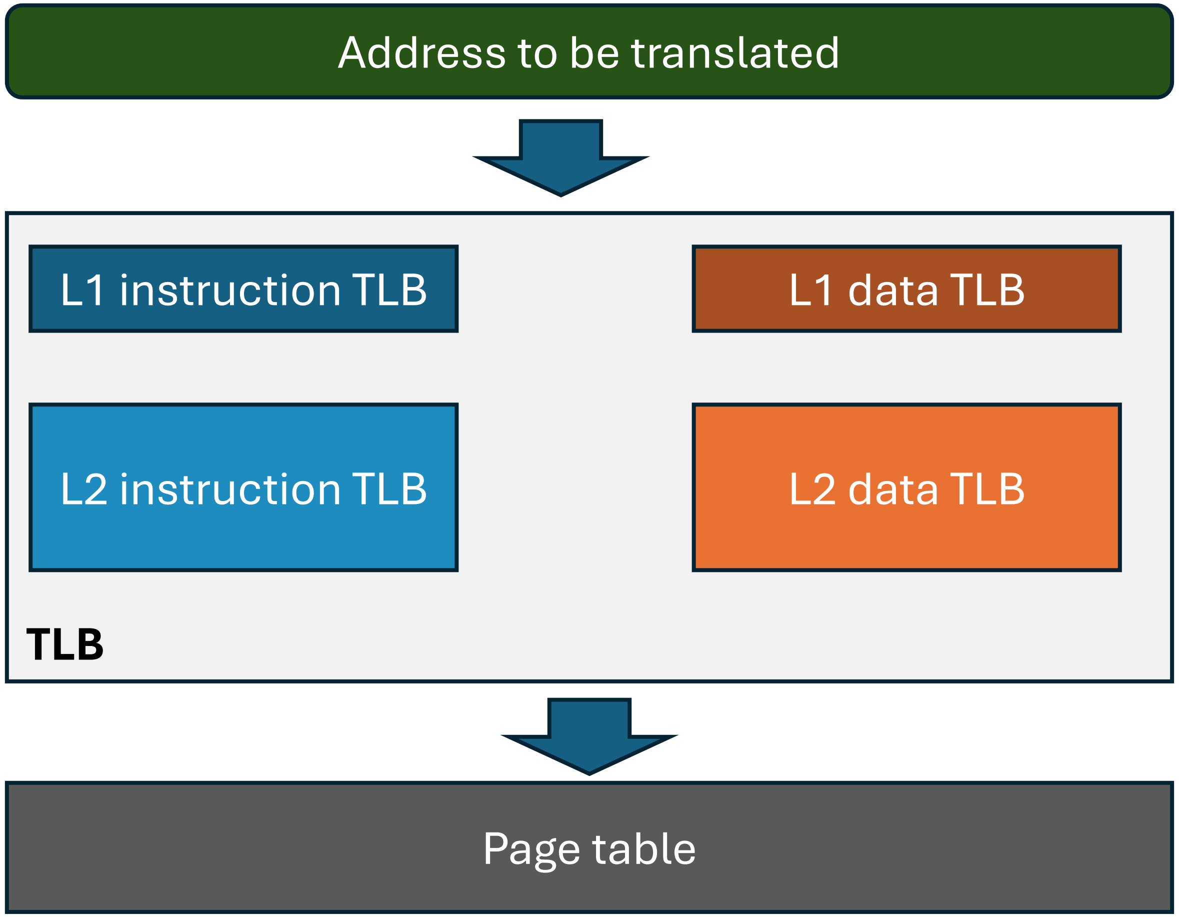
It will speculatively execute a bunch of stuff from here, including attempting to translate the user-provided address

Problem 2: observing kernel behavior

- Now we have a way to **cause the kernel to access** (however briefly) **a kernel address**
- If the provided address is **valid**, its translation **will be cached** in the TLB
- If it is **not valid**, the translation **will fail** and the TLB will remain as it is
- **But how do we know which event has happened?**

More on the TLB

- General idea: performance measurements to know when caching has occurred
- But… must know how the TLB is organized!



Reverse-engineering the TLB

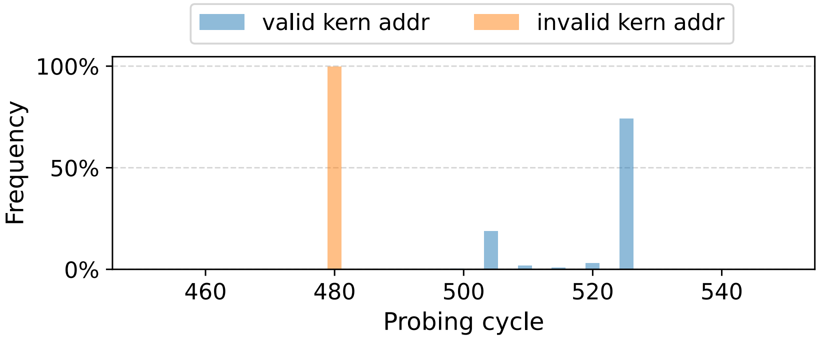
- **It gets complicated!**
- Must determine:
  - Whether **data** and **instructions** are **cached separately**
  - How many **level of caching** there are
  - **Cache parameters**: mapping, associativity
- The short of it:
  - Craft a **pattern of memory accesses** to **fill a certain number of elements** in the TLB
  - Perform **more accesses**
  - Observe whether any of the previously cached translation was **evicted**
- Do this with many different patterns and you can **estimate the TLB structure**

We reverse-engineered the TLB, now what?

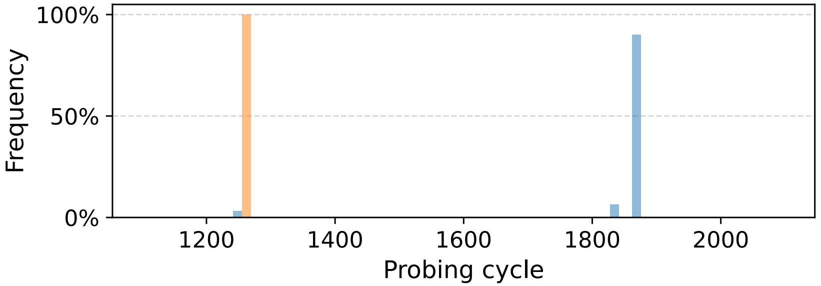
- Now we can cause the kernel to **speculatively access/try to cache** an **arbitrary address**, and check whether it was **cached**
  1. **Train branch prediction** to expect the “if” condition to be false (so it will **speculatively attempt** to **access address of interest**)
  2. **Fill data TLB** with **translations** which **compete with the address of interest** (i.e., they are allocated to the same TLB entry)
  3. **Feed** the **address of interest** to the **kernel**
  4. **Check whether the translations have been evicted** or not (i.e., measure **latency** of memory access which needs that translation):
     1. If latency is **high** -> address is **valid** (entry was evicted)
     2. If latency is **low** -> address is **invalid** (entry was not evicted)

Does it work? Yes!

- From the paper:



(a) L1 dTLB



(b) L2 dTLB

Figure 6: Measurement for two kernel addresses on the M1 CPU.

Problem 3: how do I use this to break ASLR?

- In general, **addresses where the kernel binary is stored** will be **valid**…
- … and **addresses with no kernel** will be **invalid**
- Basically, I need to find either **where the kernel begins**, or **ends**
- I can do so by **probing lots of addres**ses, and figuring out either the **lowest** or the **highest valid address**

Sample measurement result

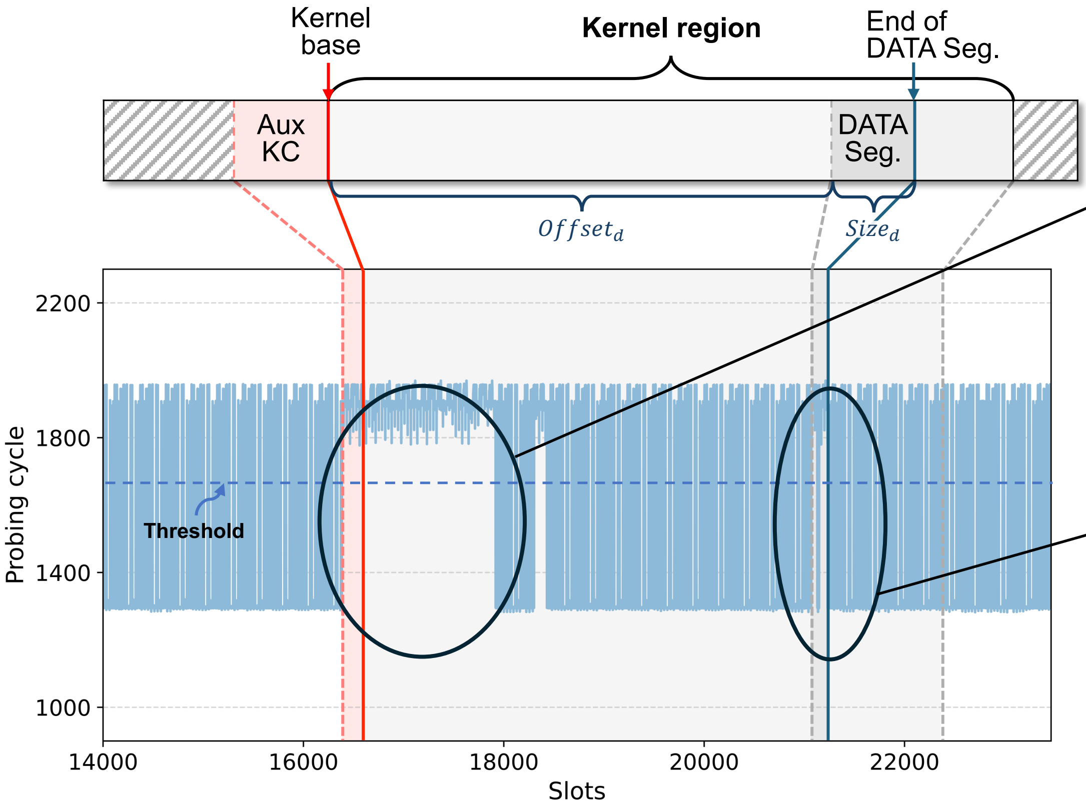


Figure 8: Probing with the attack primitive over the kernel base range.

Valid but unreliable

Valid and reliable

In summary…

- This attack **cannot reliably determine** the **lowest kernel address**
  - The kernel begins with an allocation for **third-party extensions**, which are **machine- and configuration-dependent**
  - Thus, there is a **variable valid “gap”** before the actual kernel memory
- However, the attack can reliably determine where the kernel **ends**
- It is also possible to determine **how big the kernel is** by analyzing the kernel binary (it is just a file)
- Find the end of the kernel memory region, subtract the size of the kernel image, **find the base address**
- **GAME OVER!**

There is a lot of complexity we have not discussed

- Measurements are **difficult** and **noisy**!
- **Reverse-engineering TLB structure** is **non-trivial** and a remarkable achievement in itself
- Figuring out whether the **data TLB is shared** between kernel and user space was also necessary

Possible mitigations

- **Reorder instructions** to prevent speculative execution from accessing attacker-controlled access
- Cause TLB to **allocate entries** even if **address is invalid**
- Use **fence instructions** to prevent speculative execution around memory addresses
- **Use separate TLB entries** for kernel and user space

That’s all for today!

See you in the next lecture