Meltdown Theory:

- The memory is divided into Kernel & User Space, and there is isolation between the two.
- Every Address in the Kernel Space is mapped to a valid Physical Address, as the entire Physical Space is mapped to the Kernel Space.
- Every Page in the User Space has the Entire Kernel Space mapped to it.
- A Process (in User Space) cannot directly access memory address or contents inside the Kernel Space. The Permission (permission bit) is checked to see if the process has access to a particular address.
- The Cache stores the contents of recently executed instructions for further use & accessing these elements is relatively faster (timing difference, which is exploited to infer the 'byte' values of the secret data).

Meltdown Assumption:

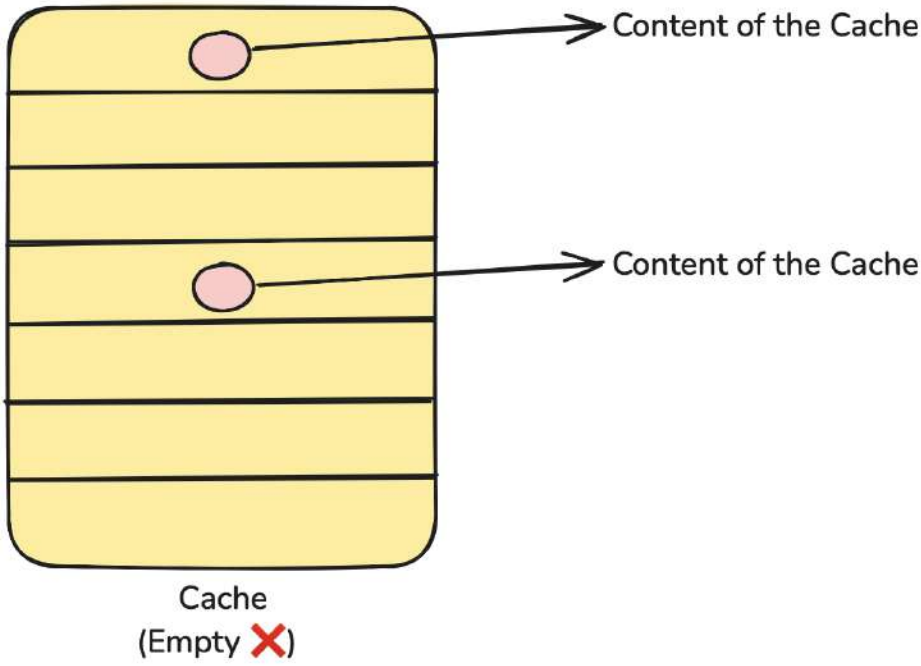
- There is no vulnerability in the OS, as the attack is targeted at the hardware surface.
- Mitigations are OFF, ie. No software based defenses.
  No Kernel Address Space Layout Randomisation (KASLR absent/disabled).

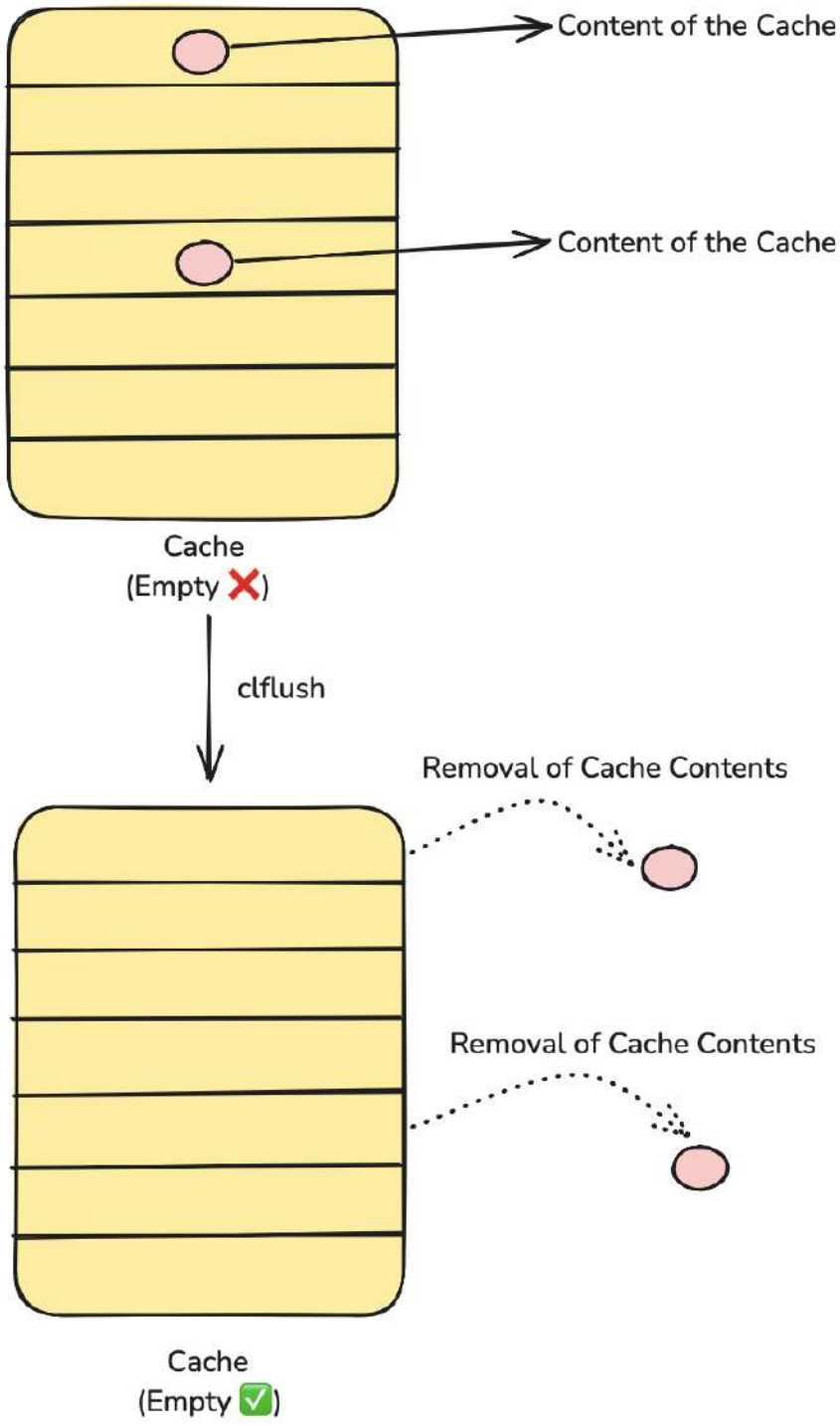
Brief Theory: KASLR (Kernel Address Space Layout Randomisation) randomises the offset of the Kernel Space & its drivers at every boot, which makes it harder for meltdown to execute, although the randomisation is limited to at max 40 bits.

Meltdown Overview:

1. Flushing of the Cache

The contents of the cache is flushed before the execution of transient instructions (speculative execution) to ensure that the cache is initially empty.
The 'clflush' instruction is used to empty the cache.
The code calling for flushing of the cache is present inside the User Space.




Content of the Cache
Content of the Cache
Cache
(Empty ❌)
clflush
Removal of Cache Contents
Removal of Cache Contents
Cache
(Empty ✅)

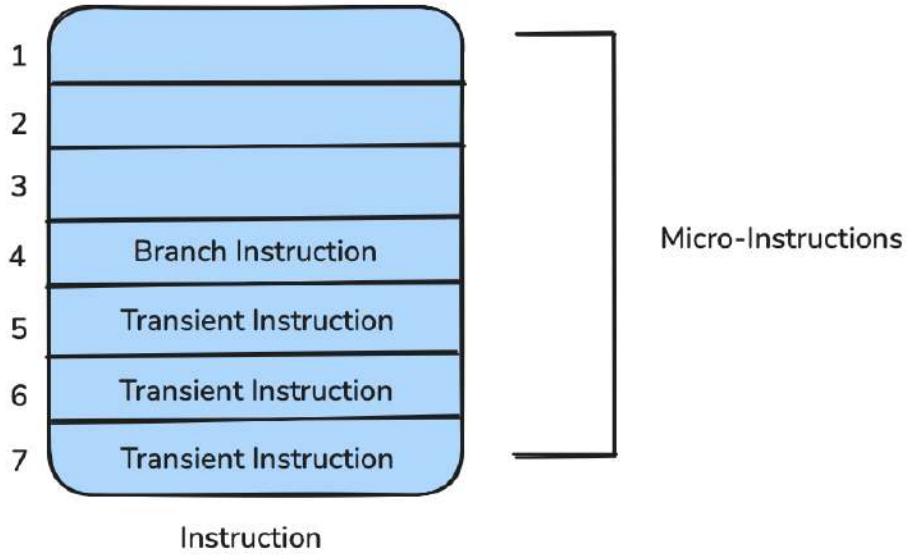
2. Speculative Execution of Transient Instructions (Out of Order Execution)

- Once the cache is empty, the speculative execution begins from a call from the code within the User Space.
- The User Space code intends to read/access the value at a chosen address(picked by the attacker) which is present inside the Kernel Space (Kernel Address).
- Using this value, an attempt is made to access an index within the Probe Array.

Quick Definitions:

1. Probe Array: The Probe Array is an array defined inside the User Space, and it must be shared (be accessible) to both the transient instructions & the flush&reload instructions, and no parts of this array should be cached (initially).
   The size of the probe is 256*4096.
   The value of 256 accounts to all types of permutations of a byte(2^8=256) & 4096 (4KB) is accommodated to map it to the size of a user space page.

2. Instruction has 3 phases:
   (a) Fetching (Loading & Decoding of the Instruction)
   (b) Execution (Converting the Instruction into Micro Operations & executing them)
   (c) Retiring (Checking Permission & Commiting the State)



The CPU predicts the result of the Branch Instruction (Branch Predictor) in advance & begins executing the following instructions (out of order execution), effecting both the register and cache states as the values are loaded in both these structures.

Note: Instructions having no dependencies (independent values used in execution & no wait for a result from previous instructions) are executed & retired in-order.

The Branch Instruction is that of accessing the memory address (in kernel space), whose permission would be checked using the permission bit.
But since the CPU begins out-of-order execution of the following instructions for performance reasons, there is a race condition between the result of the permission check & the execution of the transient instructions.

If the prediction of the Branch Instruction comes out to be false (the CPU assumes it to be True initially, to begin speculative execution), the register state gets rollbacked to the previous state but NOT the Cache. Hence, speculative execution leaves a trace (affects the microarchitectural state).

There are two ways to handle this race condition:

1. Exception Handling: The process used here is to fork the attacking process where the child accesses the kernel memory location before crashing & the parent process continues the the remaining code (recover the secret).
2. Exception Suppression: Memory Accesses are grouped to an atomic operation, and has an option to rollback to a previous state if an error (crash) occurs.

The Byte value located at the Target Kernel Address is stored in the RCX register.
ONE Byte is loaded into the LSB of the RAX register (denoted by Al).

Note: 'Al' denotes the Least Significant Bits (8 bits) of the RAX register.

Why only 1 Byte is Loaded?
- For performance & efficiency of the attack.
- Transmission/Leakage of secrets in meltdown occurs in a byte-by-byte manner.
- Another Faster (not foolproof) method is transmission in bits (bit-by-bit leakage of secret).

There are two ways to handle this race condition:

1. Exception Handling: The process used here is to fork the attacking process where the child accesses the kernel memory location before crashing & the parent process continues the the remaining code (recover the secret).
2. Exception Suppression: Memory Accesses are grouped to an atomic operation, and has an option to rollback to a previous state if an error (crash) occurs.

The Byte value located at the Target Kernel Address is stored in the RCX register.
ONE Byte is loaded into the LSB of the RAX register (denoted by Al).

Note: 'Al' denotes the Least Significant Bits (8 bits) of the RAX register.

Why only 1 Byte is Loaded?
- For performance & efficiency of the attack.
- Transmission/Leakage of secrets in meltdown occurs in a byte-by-byte manner.
- Another Faster (not foolproof) method is transmission in bits (bit-by-bit leakage of secret).

This value, say 'x', loaded into the Al register, is now multiplied by 4KB (4096), to ensure that the hardware prefetcher does not load adjacent memory locations into the cache.

x: Value at Target Kernel Address

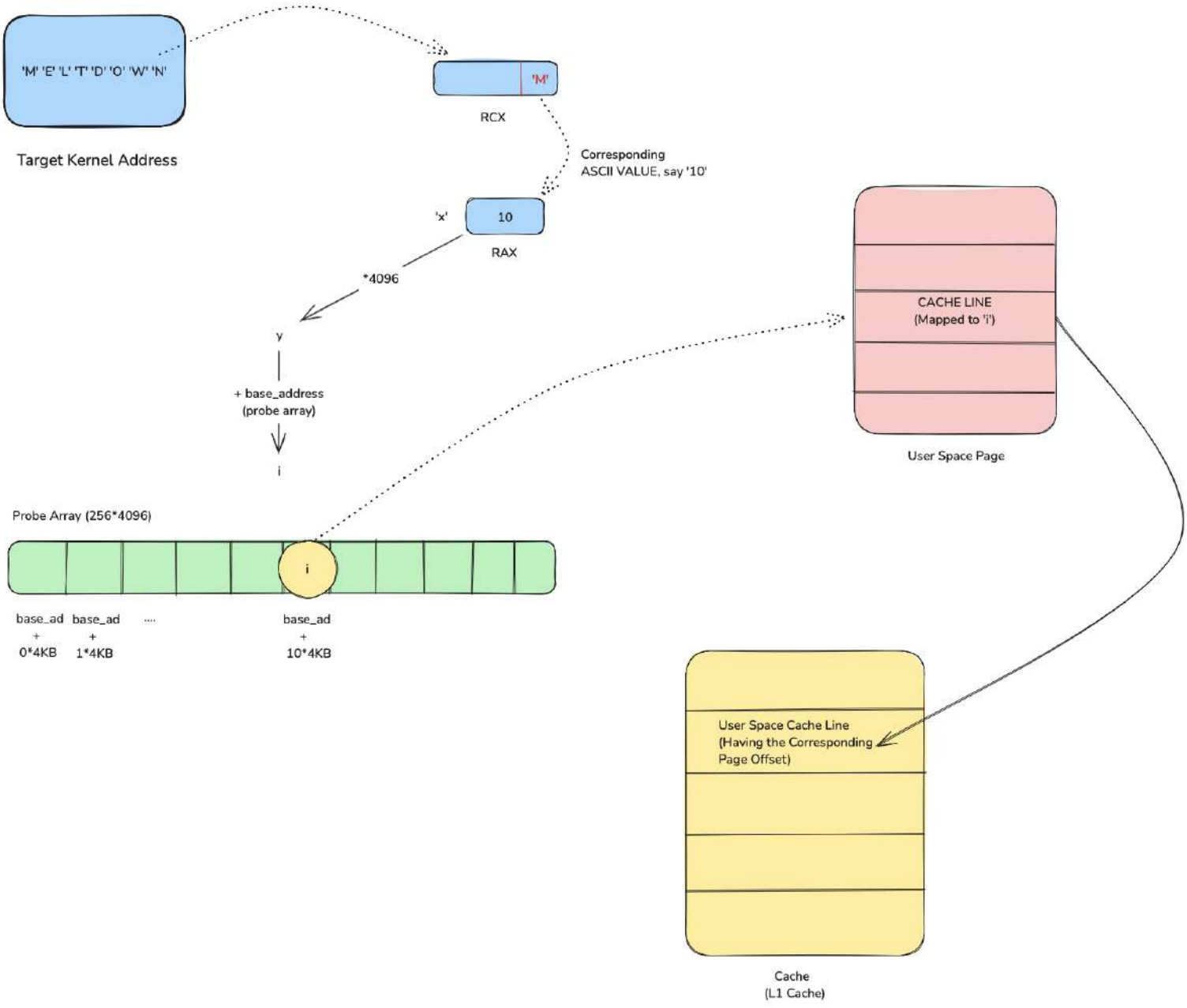
Now this value, say 'y', is added to the base address of the probe array, and the result of this addition gives the index of access to the probe.

y: x*4096

i = base_address(probe array) + y

The cache line corresponding to this address is cached (L1 cache of the requesting core).

Note: There's an inherent bias towards a value of Zero when the contents of the target address is not available, thus, there's a retry micro-instruction that retries the access a certain number of times.



Once, the speculative execution is complete, the cache stores the corresponding cache line mapped to the index 'i'.

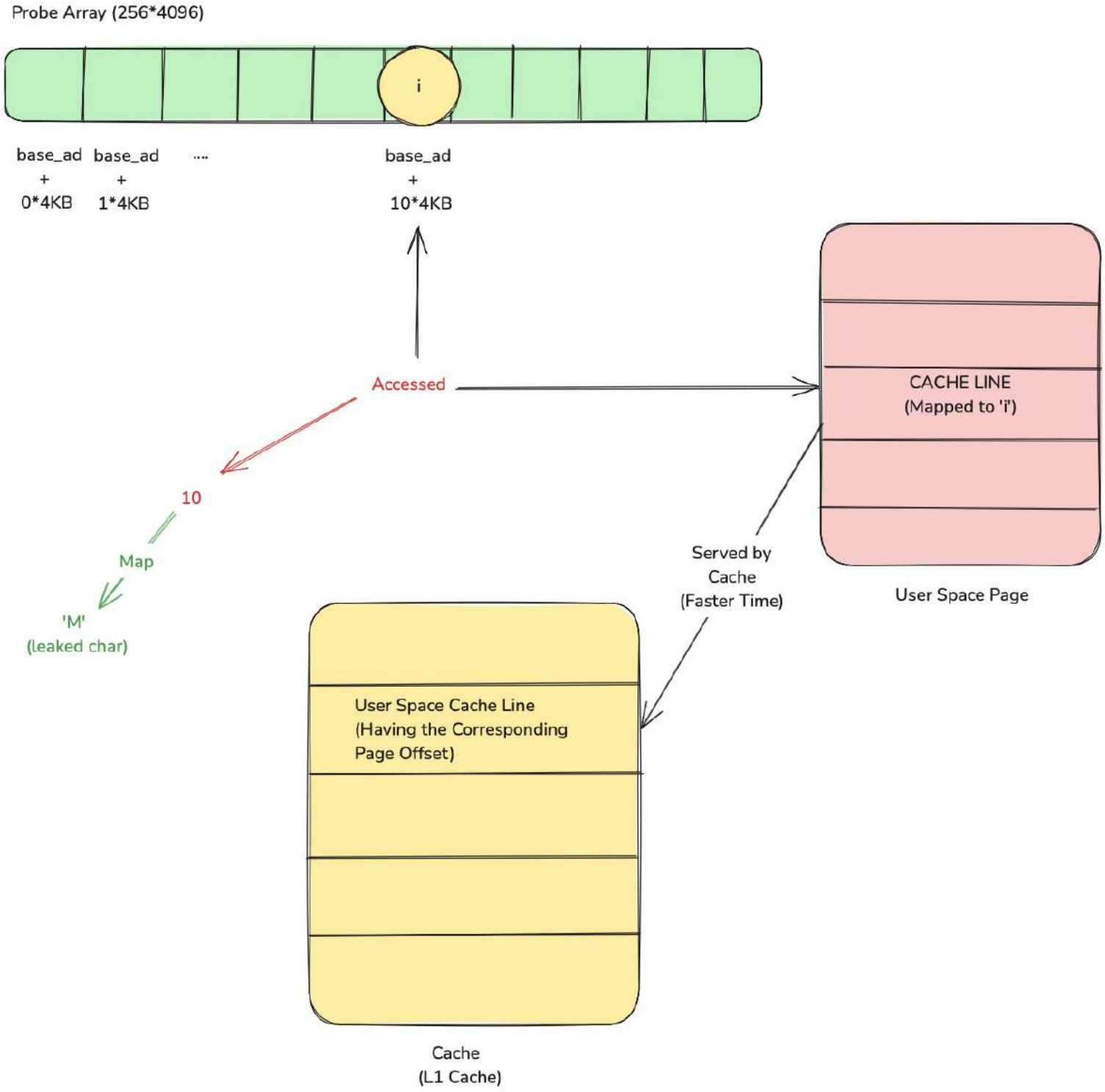
3. Reload (Reloading the Content of the Probe Array)

- The Elements of the Probe Array are iteratively accessed, i.e., the time of access to each element of the probe array is measured using the 'rdtsc' instruction to see which part of the array was cached during the speculative execution.

- The iterations happens in a span/gap of 4096 (4KB), i.e., the Page offsets are checked to see what part was cached during the speculative execution.

- Since, 'i' (base_addr + 10*4096) was accessed during the speculative execution, the corresponding cache line to this index of the probe will be present inside the cache & the time of access will be relatively fast/lower in comparison to other elements of the probe, thus inferring the value (10, which hypothetically represented 'M' as ASCII), and the secret will be leaked byte-by-byte (character by character).

Note: The value of 'x' can be anything from 0-255, and thus measuring access times to every part of the probe will take at max 256 iterations. Thus, the secret byte can be inferred in at max 256 iterations (in this case, it will be discovered in 11 iterations)


Probe Array (256*4096)
i
base_ad + 0*4KB
base_ad + 1*4KB
....
base_ad + 10*4KB
Accessed
CACHE LINE (Mapped to 'i')
10
Map
'M' (leaked char)
Served by Cache (Faster Time)
User Space Page
User Space Cache Line (Having the Corresponding Page Offset)
Cache (L1 Cache)