内存隔离

内存隔离是当今操作系统的一个核心安全特性。操作系统确保用户程序无法访问彼此的内存或内核内存。

这种隔离是我们计算环境的基石，它允许在个人设备上同时运行多个应用程序，或在云服务器上执行多个用户的进程。

每个虚拟地址空间本身被划分为一个用户和一个内核部分。虽然运行的应用程序可以访问用户地址空间，但只有在CPU以特权模式运行时才能访问内核地址空间。

有一个特定 bit 位来标记一个内核空间中的内存页是否可以被访问，当处于内核态时记为1，表示可以被访问。在用户态时，记为0，表示不可以被访问。

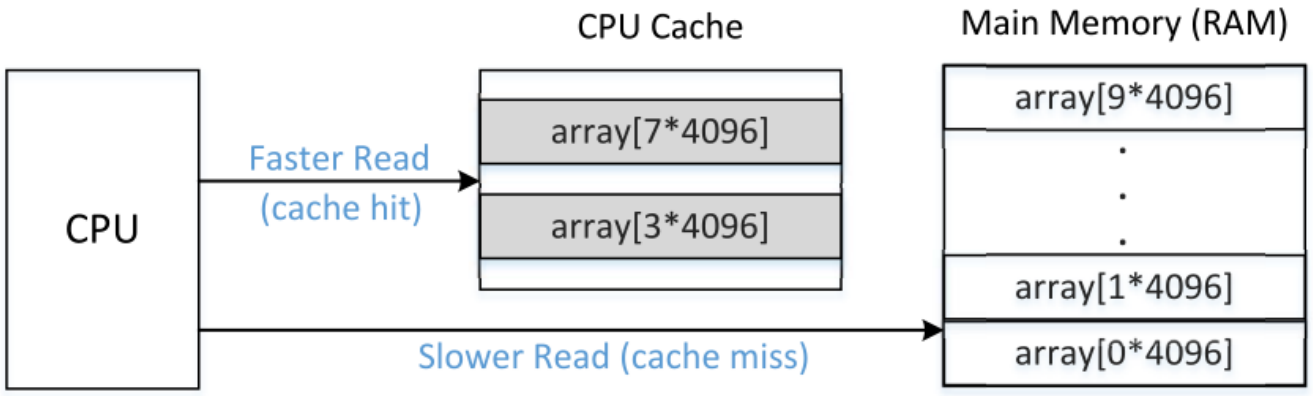
而Meltdown所作的事情，就是打破这种内存隔离。它为用户进程提供了一种很简单的方式来获取其执行机器的整个内核内存，以及映射在内核区域中所有的物理内存。

Cache侧通道

Cache是介于CPU和主存储器之间的高速小容量存储器。它基于程序的局部性原理，缓解了高速CPU和 慢速主存之间的速度差异。

从CPU缓存访问数据比从主内存访问数据快得多。当数据从主内存中提取时，它们通常就会同时被调入Cache中，因此如果再次使用相同的数据，访问时间将快得多。

因此，当CPU需要访问某些数据时，它首先查看其缓存。如果数据在那里（这称为缓存命中），它将直接从那里获取。如果数据不在那里（这称为未命中），CPU将转到主内存以获取数据。在后一种情况下花费的时间要长得多。



乱序执行

乱序执行是一种优化技术，它允许CPU尽可能充分地利用CPU核心的所有执行单元

支持乱序执行的CPU，不是严格按照顺序执行处理指令，而是在所需资源可用时立即执行指令。

从安全性的角度来看，有一点特别重要：一般的易受攻击的乱序执行CPU允许无特权进程，将数据从特权地址加载到临时CPU寄存器。

此外，CPU甚至基于该寄存器的值进行了进一步的计算。例如，我们通过该寄存器的值来访问数组。尽管，在执行权限检查之后，我们访问数组的值被丢弃，程序被回滚，寄存器被清空。

然而，我们发现乱序执行虽然消除了对寄存器和内存的影响，但是没有消除对 Cache 的影响。

简单的例子

```
raise_exception()
// the line below is never reached
access(probe_array[data*4096])
```

站在我们的视角,程序按顺序执行,在抛出异常之后,操作系统会执行陷入指令并终止应用程序的执行。但是，由于乱序执行，如果下面的访问数组的指令，不依赖于上面的触发异常的指令的话，那么CPU很可能已经执行了下面的访问数组的指令。

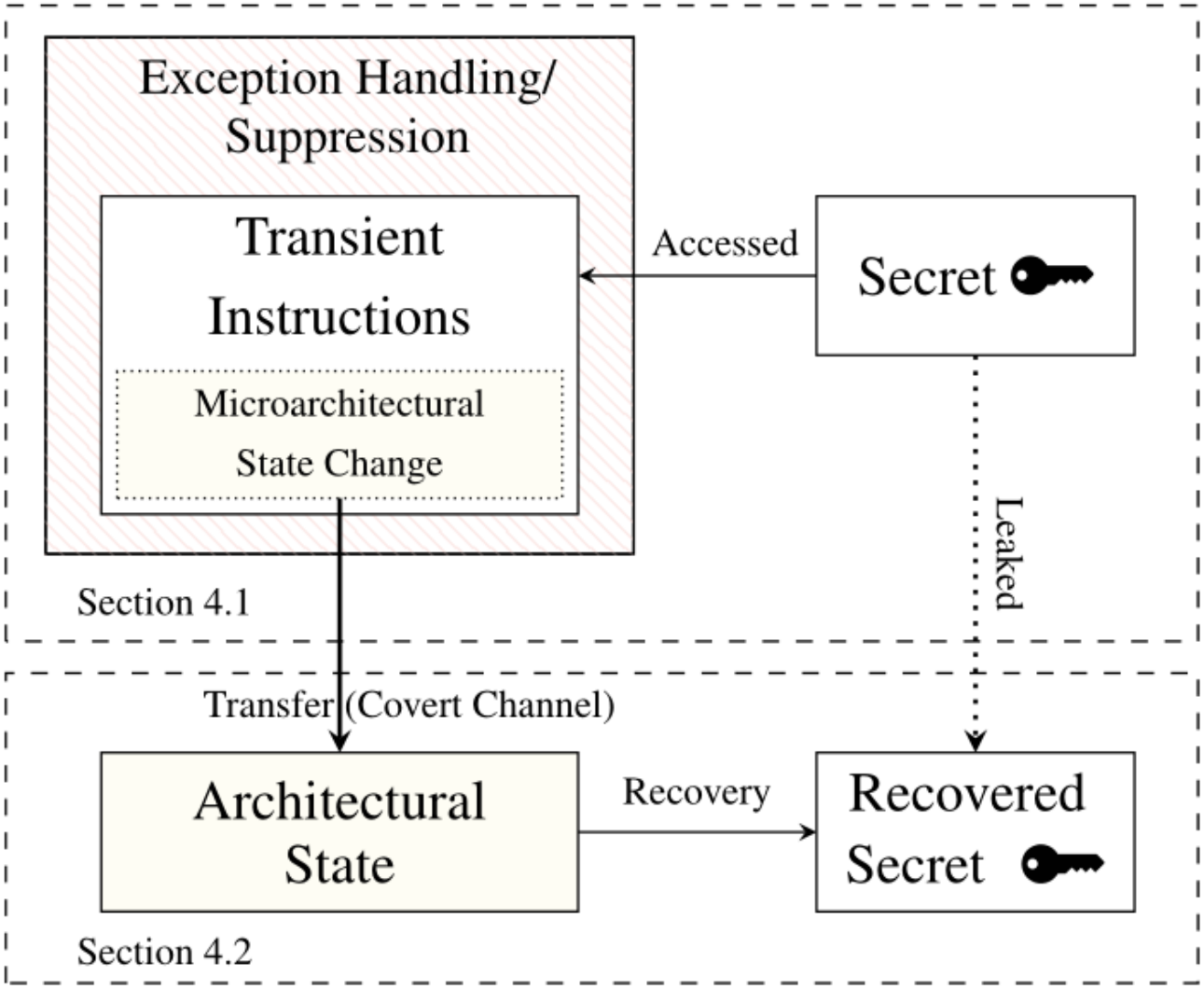
尽管这样的执行，从宏观来看没有任何的影响。因为在异常被检测到之后，之前的运行结果都会被回滚。但是，这个数组的数据已经被缓存到 Cache 中了。

这里*4096，是为了让每一个数组元素，落在不同的Cacheline上，因为如果他们在同一个Cacheline中，会降低攻击的精度。

Meltdown

Meltdown攻击分为两个攻击模块

- 第一个模块是让CPU执行一条或多条在执行路径中永远不会发生的指令。
  在上面的简单例子中，就是访问数组的语句。
  这种指令我们称为**瞬态指令**
- 第二个模块是建立Cache侧信道，将瞬态指令序列对体系结构的影响转移出来



执行瞬态指令

访问用户无法访问的地址，会触发异常，通常会终止用户程序。

所以我们必须处理此异常，有两种方法：

- 捕获异常
- 异常抑制

异常处理

显然，我们可以定义自己的异常信号处理程序，该处理程序在发生特定异常时执行。

它允许攻击者发出指令序列，并防止应用程序崩溃。

异常抑制

处理异常的另一种方法是在第一时间阻止他们抛出。

事务性内存允许将内存访问划分成为一个伪原子操作，如果发生错误，可以选择回滚到以前的状态。如果在事务中发生异常，架构状态将被重置，程序继续运行而不会中断。

此外，由于分支预测失误，推测性执行发出的指令可能不会出现在执行的代码路径上。依赖于前面的条件分支的此类指令可以推测性地执行。

因此，无效的内存访问被放在推测指令序列中，该指令序列仅在先前的分支条件计算为true时执行。通过确保条件在执行的代码路径中永远不会计算为true，我们可以抑制发生的异常，因为内存访问只是推测性地执行。

构建侧信道

我们用之前提到过的 FLUSH + RELOAD 的方式来构建侧信道。因为它可以构建快速、低噪声的隐蔽侧信道。

主要分为三个步骤：

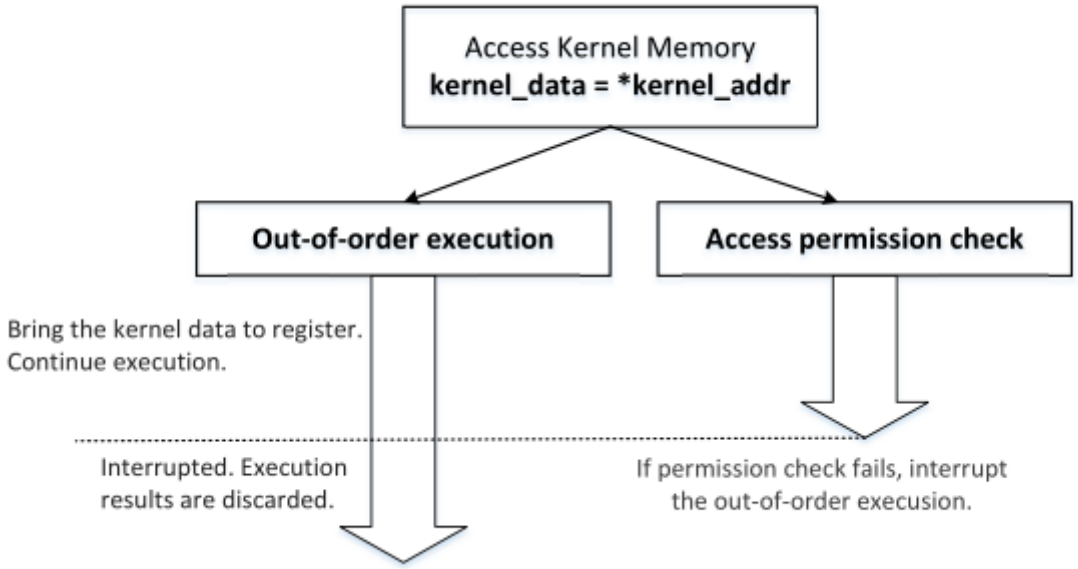
- 首先我们 FLUSH 整个 Cache空间，将目标数据从Cache逐出。
- 执行受害者程序，也就是瞬态指令序列。
- RELOAD 目标数据，观察比较访问时间，推测出 Secret 值。

Step one：读取内存中的秘密数据

访问内核地址空间时，只要创建了虚拟地址的映射，CPU都可以访问这些内容。只是内存隔离的存在，导致我们无法从用户态访问。

和访问用户地址空间不同的一点是，访问内核地址空间需要进行权限检查，也就是查看 Supervisor bit。如果权限检查失败，则会触发异常。

而现代CPU 的乱序执行特性，使得它可以在非法访问内存和引发异常之间的瞬态窗口内执行瞬态指令序列。




```
1 ; rcx = kernel address, rbx = probe array
2 xor rax, rax
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

在这里，第四行的MOV 指令试图读取内核地址中的 Secret 到寄存器 rax 。第七行的MOV 指令，将 Secret 参与运算后，放到 rbx。

第四行的 MOV 指令，分为两个基本操作，取Secret 到寄存器，同进行权限检查。**而如果 Secret 已经在Cache中，那么从Cache 加载到寄存器的速度将非常快，而权限检查则需要一定时间。这里就出现了可利用的瞬态窗口。**

Step two：传递 Secret

由于 CPU 的乱序执行，当我们将Secret 从Cache 加载到寄存器中后，此时权限检查未完成，而第七行MOV指令，需要的资源已经完备，所以它可以进入执行，我们成功访问了probe_array[secret + rbx]，它已经被缓存到Cache中。

尽管，当权限检查完成时，寄存器rbx 的内容会被回滚，但是在Cache中它依然存在。

Step three：Receiving the Secret

probe_array[secret + rbx]在缓存中时，我们用 FLUSH + RELOAD就可检测到命中时间最短的数组元素的编号 index。

那么 secret = index - rbx

优化和限制

读出为0的情况

在乱序执行加载操作期间，如果权限检查速度快于瞬态指令的执行，那么非法内存加载（第四行的MOV指令）会返回 0。如果我们不用 MOV 而是用 add 可以很清楚看到这一点。

对于这样的 0，我们设置了 Retry，直至读出不为 0 。

那么当我们的 Secret本身就是 0，那么怎么办呢?

显然，当我们访问Cache，所有probe_array[]元素之间的访问时间没有差异时，我们认为读出的 Secret为0。

单比特传输

Intel TSX 异常抑制

使用Intel TSX时，它将多条指令组合到一个事务当中，事务的定义就像是一个原子操作，即要么全部执行，要么全部不执行。

如果事务中一条指令失败，则已经执行的指令都会被恢复，但不会产生异常。

如果我们TSX 包装瞬态指令序列，那么任何异常都会被抑制。然而，微体系结构层面的状态变化依然存在。

而异常抑制比捕获异常后执行后续操作要快得多。

打破 KASLR

KASLR ，内核地址空间随即布局技术。，它允许在引导时随机布局内核代码的位置。也就是在内核映射在物理地址空间是随机的，而不是固定在某个位置，因此我们需要在攻击之前获得随即偏移量。

这个偏移量的位数限制在 40位以内。如果假设目标机器设置有 8GB 的RAM，那么以 8GB（2^32） 的步长，在最坏的情况下，只需要128次测试，就可以覆盖 40位的搜索空间。