[硬件系统辅助系统安全 1](#_Toc105005650)

[review 1](#_Toc105005651)

[侧信道和隐秘信道 2](#_Toc105005652)

[不经意随机访问内存 2](#_Toc105005653)

[case：meltdown 3](#_Toc105005654)

[当OS不再可信时…… 7](#_Toc105005655)

[安全处理器 9](#_Toc105005656)

[TEE：可信执行环境 9](#_Toc105005657)

# 硬件系统辅助系统安全

## review



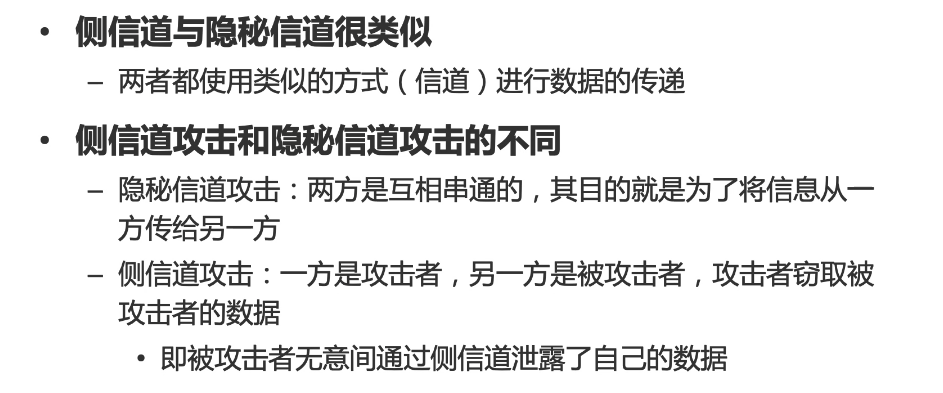
从OS原来角度来讲有DAC和MAC，建立在OS是可信的情况下。在这个情况下去建立一系列规则。其实有一个假设是不一样的，一旦我对一个Object可读可惜可执行我就是它的owner，而MAC则不一样，它认为有一个更高的存在，它来制定最高的规则。

DAC的两种做法：windows还有Posix这样9个bit。

SELinux：security enhanced linux

一旦某个文件被打上标签，说是private，它就不应该放在公网能访问的地方。比如nginx的进程被打上标签，它就不能够访问。

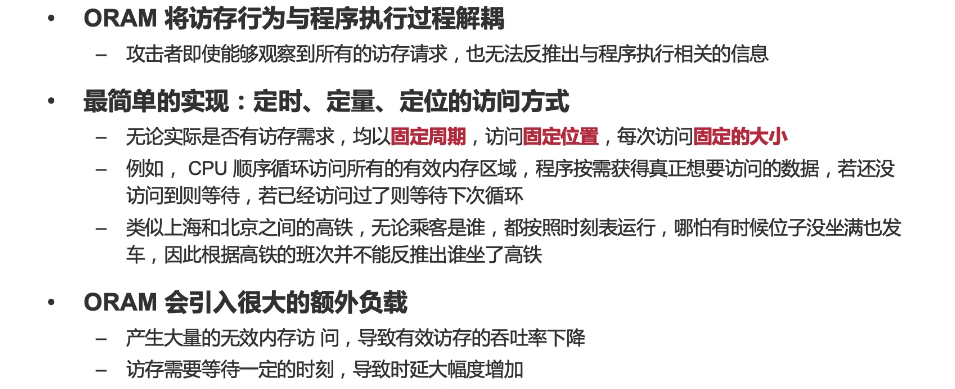
### 侧信道和隐秘信道



有没有串通。他们一个核心的点在于说通过侧信道一些本来不是用来做通信的信道被用来做侧信道。只要有共享就有可能被侧信道攻击。

timing channel：为了解决这个问题：我们提出了一个常量时间算法，不管你算什么东西，你最后都是用同样的时间。用同样的时间你就不知道走哪条路径，你就不知道secret是0还是非0。但是速度会变慢。

### 不经意随机访问内存



它将访存的行为和程序执行的过程解耦。

做一个小硬件，它下面是金手指，上面是内存插槽。这样就可以做到man in the middle。可以看到所有内存的读和写。我可以第内存加密，你看到也不知道我在干嘛。但我可以通过看到你访问哪些地方的数据，访问了多大的数据，然后知道你在干什么。

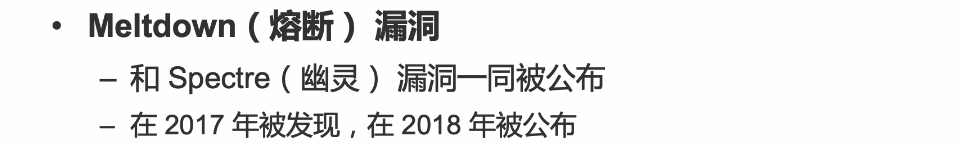
e.g.：税表：这个文件可能0-10w，10w-30w大小不一样，我就可以通过看到访问范围来推测出你的工资，最好的方法是每个人的表一样。最简单的实现：不管什么样的内存访问，它都是一样的。只要内存访问一点点不一样，它就可能会变成一个side channel。但怎么可能所有人都一样呢？这里一个point：规定一个访问的pattern。应用程序逻辑要符合我这个pattern。

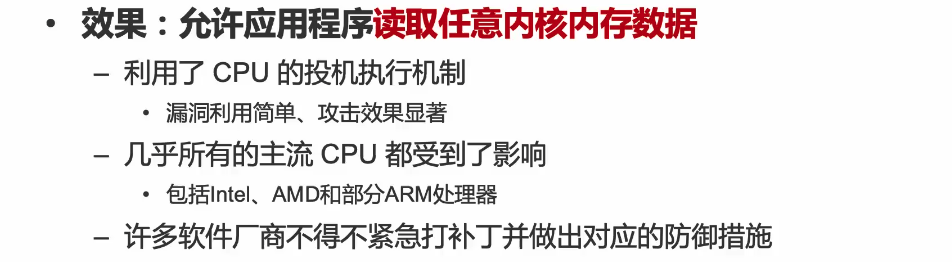
比如内存访问每一次都全部扫描：你要读的话顺便在其中读想要的bit。毫无疑问会引入大量的额外负载。原来是你只访问你需要的，现在要变成一样的，就要求一个最小公共集合（很可能就变成了整个memory）。时延上升，吞吐量下降。ORAM非常昂贵，对于安全非常敏感的场景还是有人用的。

## case：meltdown

曾几何时觉得side channel不那么关键。首先内存memory pattern很难看到，这要改硬件。机箱在我家里谁能改啊。真实世界有很多噪音，哪怕你运行两次同样的代码时间都有可能不同。攻击者怎么测呢？它还要恰好在运行在if的时候去攻击cache。对于安全critical，运行东西比较简单的场景才要考虑。

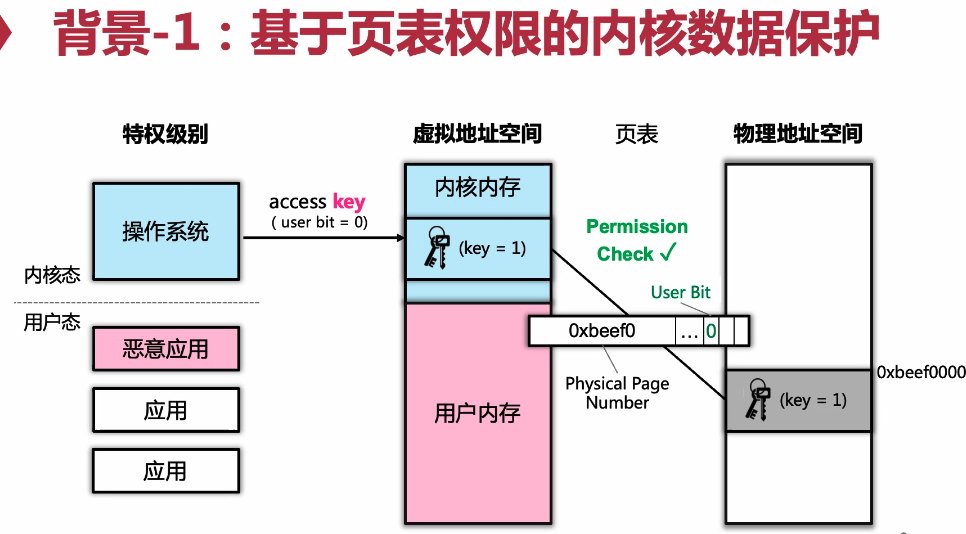
18年的时候meltdown一出来，把整个工业界和学术界都震动了。



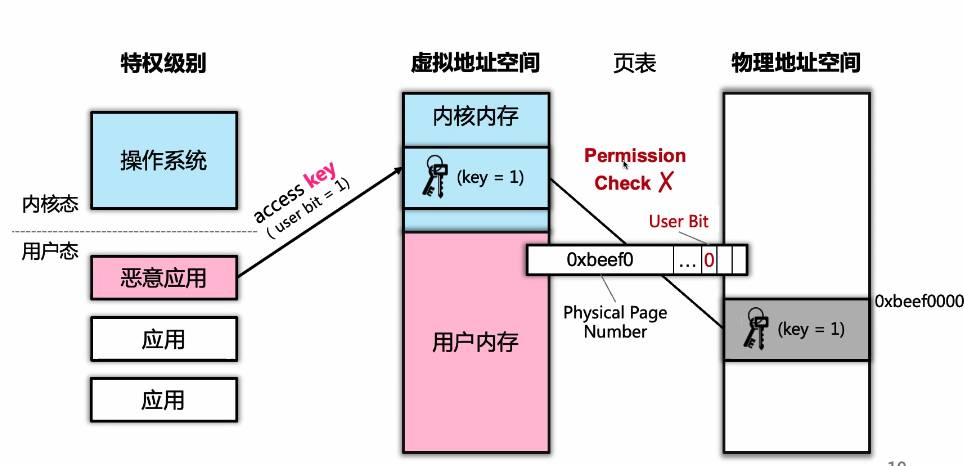


内核有一段是来做direct mapping，意思是把所有的physical memory都映射到自己的一段虚拟内存，这就会使得所有云厂商崩溃，一个人能看到所有人的数据，那谁还允许别人和自己运行在同一台物理机上。

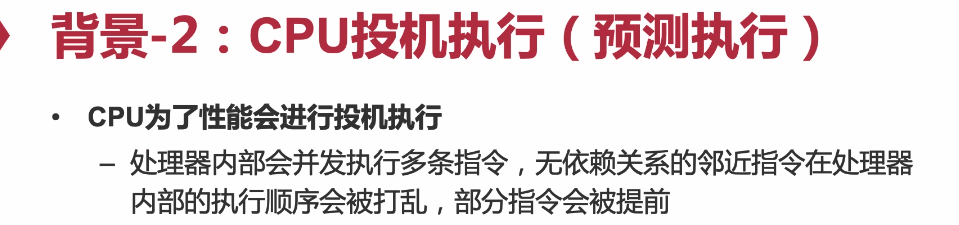
投机执行：做的越好的CPU越有。比如简单的比如MCU，RISC-V没有。



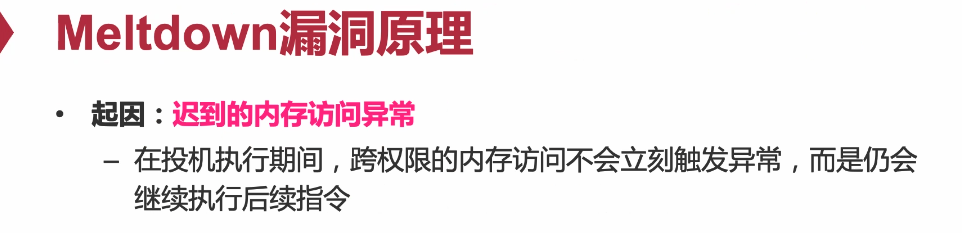
key是一个关键数据，它在虚拟内存上和物理内存上都有一块。



user bit是1，MMU就会禁止访问。发生一个trap到内核去处理。meltdown是用户发起了一个请求之后用户还能去读这个数据。



提前如果执行错了怎么办，比如一条if指令条件还没算出来后面那条指令还要不要做，先把if后面那条指令load进来，一条bne和一条if同时做，如果if是false，后面的指令就回滚掉。如果if是错的，后面的就白做了，白做就白做了，只是花一点电费，反正闲着也是闲着。提到回滚：如果执行对了就对了，如果一个CPU很复杂能不能保证所有状态都会滚。in fact：it cannot。



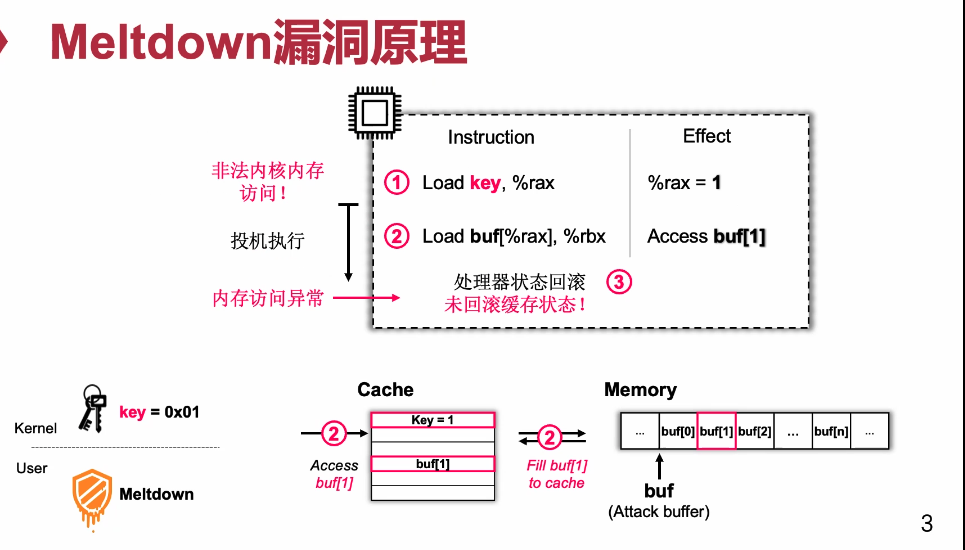
投机执行期间，跨权限的访问不会触发异常，但这个执行不会commit。

现在用户态访问内核态地址，直接执行会报错，如果这条指令是在一条if的后面，然后这个if最终会变成false，于是CPU会去执行这条访存指令，比如cache会发生变化。然后if结果出来，是个false，就会把这条指令回滚掉，但是不会报错，我没执行为啥会报错啊。但是cache的状态发生了变化，cache回滚非常复杂。比如cache发生了一个evict，你回滚怎么回滚？你只能把evict的东西先记在一个地方，你要记多少呢？记在哪里呢？

要增加很多硬件的设备，资源。



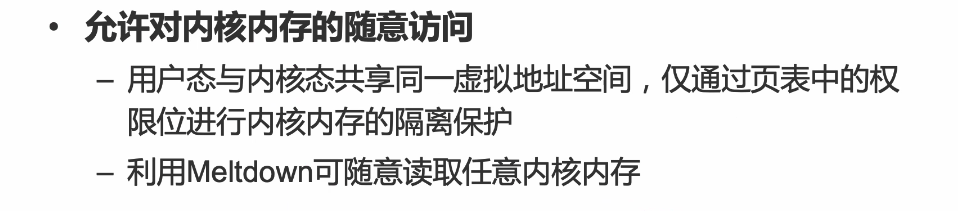
我们可以通过cache知道你做了什么事情。根据cache的变化推测出执行了哪条指令。



把要偷的这个值变成一个地址，然后它就去访问buf[1]这个地方，它根据cache的映射的话它会映射到某一块cache里面，然后这个时候内存访问异常，然后回滚。CPU状态回滚。但是没有把buf[1]从cache里删掉。我们可以通过没有回滚的cache来窃取目标数据。最后的结果：要么就是buf[1]被填上了要么就是buf[0]被填上了。假设这个key要么是1要么是0，由于我的代码访问了buf[1]那么我的cache要么是buf[1]要么是buf[0]被填上了。

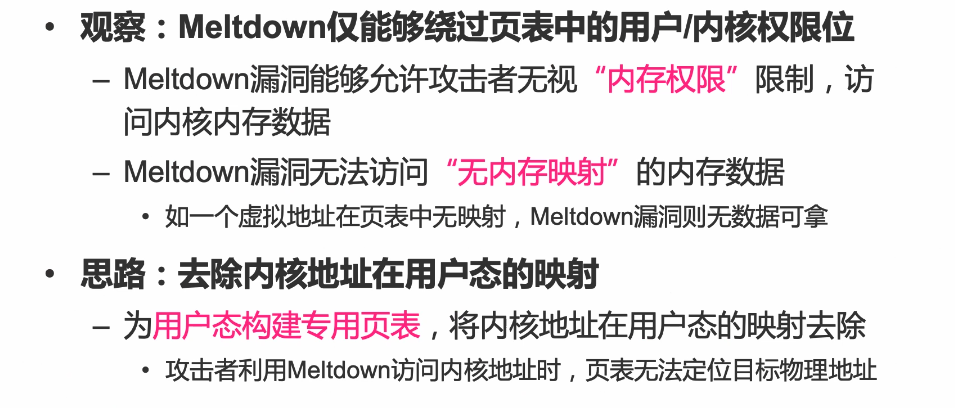
全相连可能看不出来，但是现在都是组相连，buf[0]和buf[1]就会被落在不同的组，你只要在组这个粒度被判断哪个被改了，就知道key是多少。因为整个cache对应用程序来说是和内核共享的。

由于用户态和内核态共享同一个地址空间。

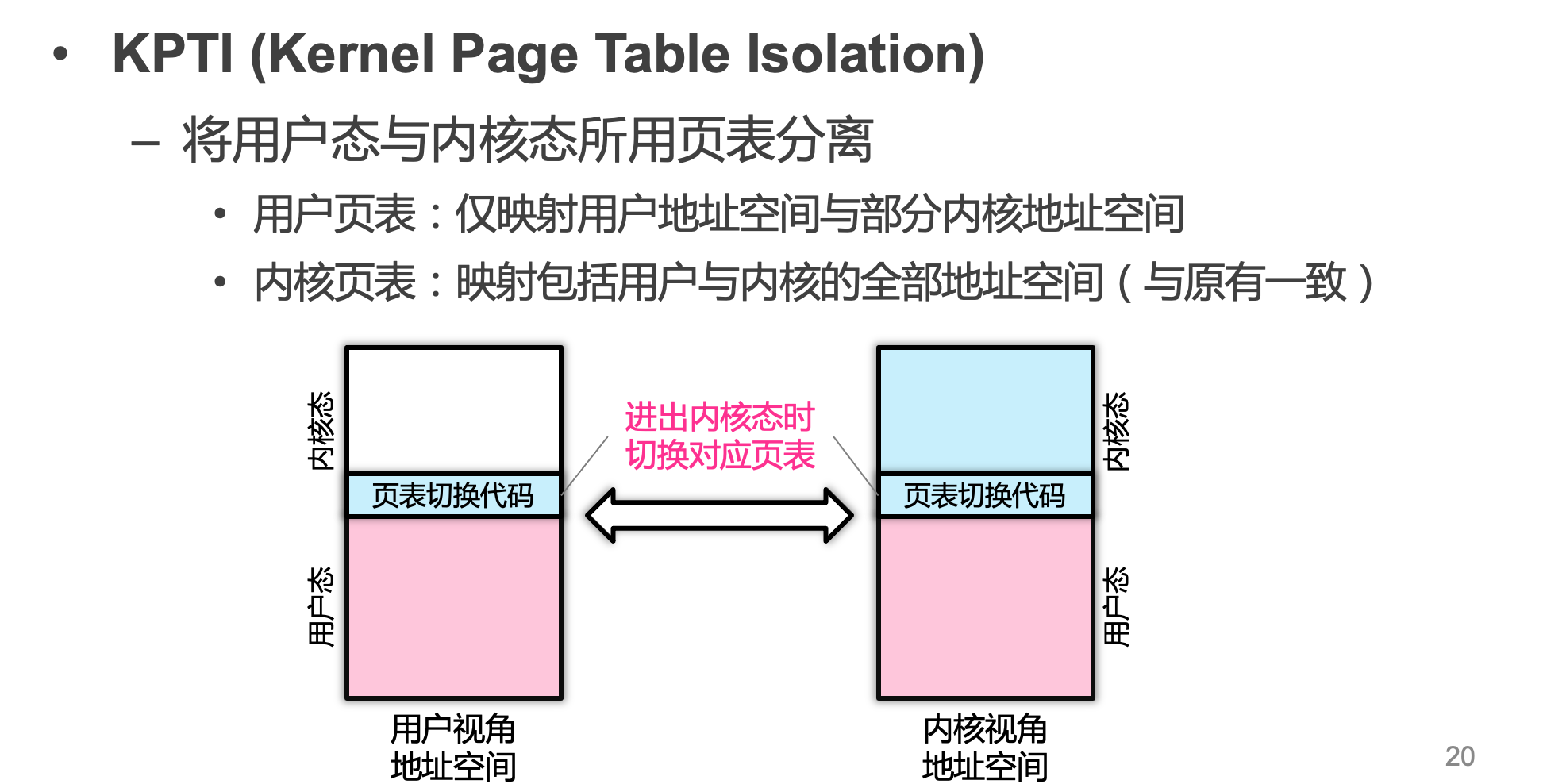


代码写得好一点的话，一秒钟可以投几百k。还是非常危险的。这种攻击叫：covert channel。这个串通体现在什么地方呢？这两条指令就是串通的人。（差2min）

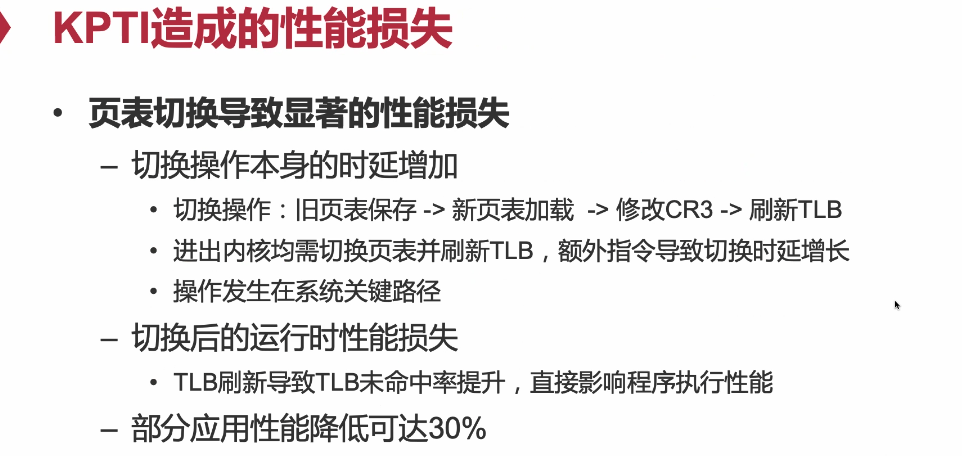
它们写任何东西都会rollback除了cache。



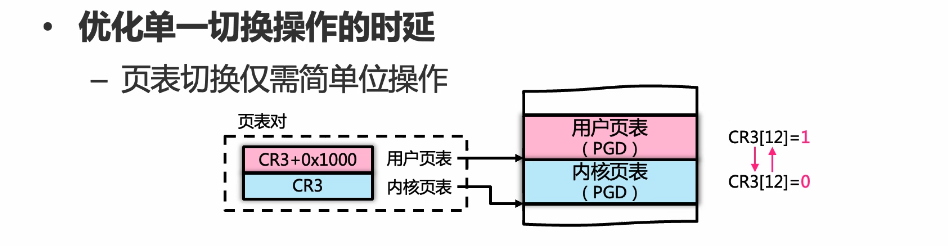
用户态为什么能知道内核态的地址。传统OS内核和用户态在一个地址空间，内核在高位，用户态在低位。

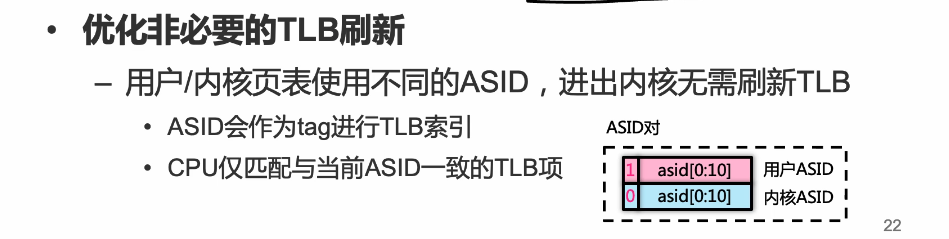


两张页表，但会导致性能损失。



性能优化：





就算是这样，还会有比较大的overhead。linus不喜欢这个patch，毕竟它把性能降低了。

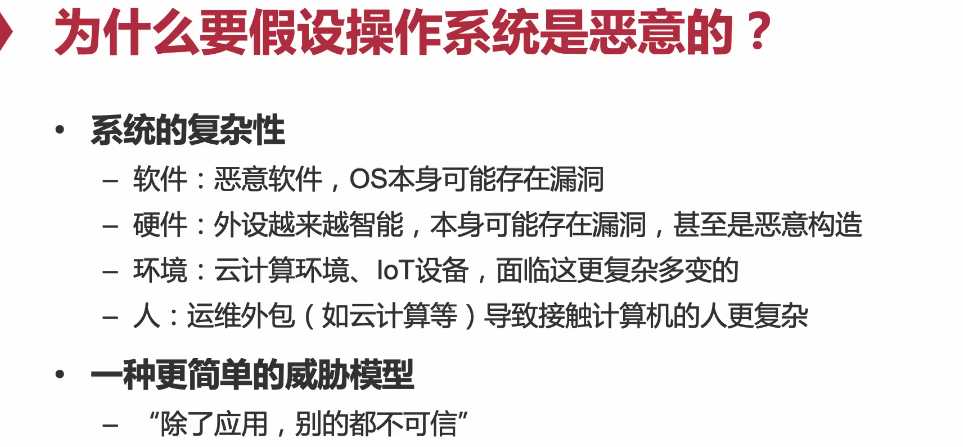
现在很多服务器购买了新的CPU，在乱序的时候也去检查exception。但还是有很多的老的CPU。大家可以尝试一下。



spectre同样也是利用乱序执行这100个cycle去做一些事情。

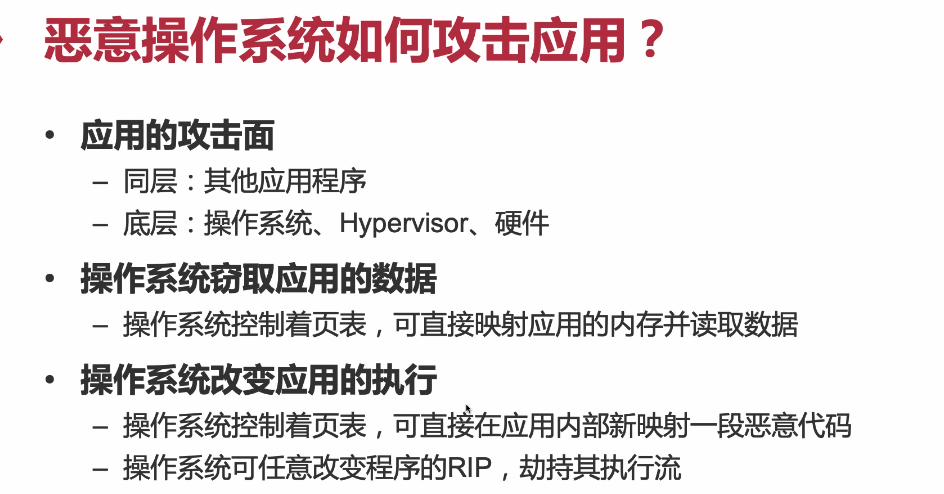
## 当OS不再可信时……

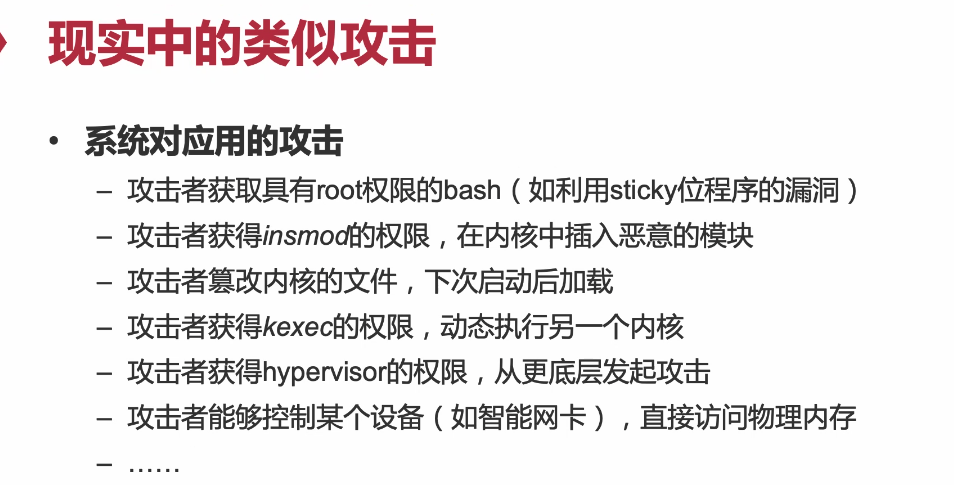
我不信OS我装它干啥呢？



直到今天还有很多人在讨论随机执行体系结构怎么办？从meltdown到今天都四年了。

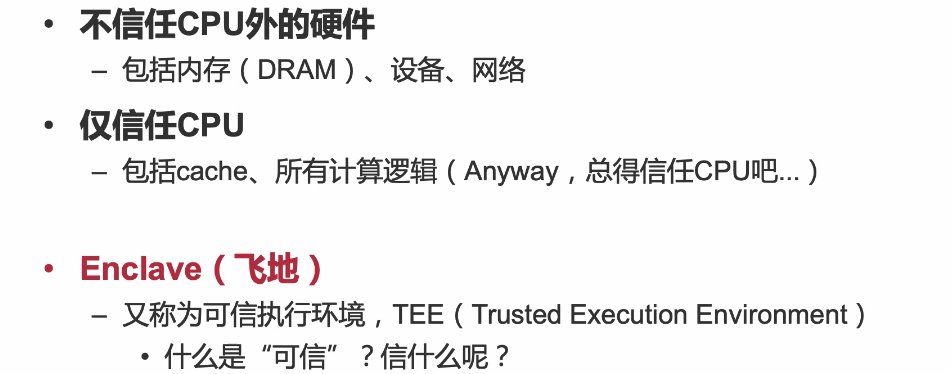
人是安全模型里最危险的。



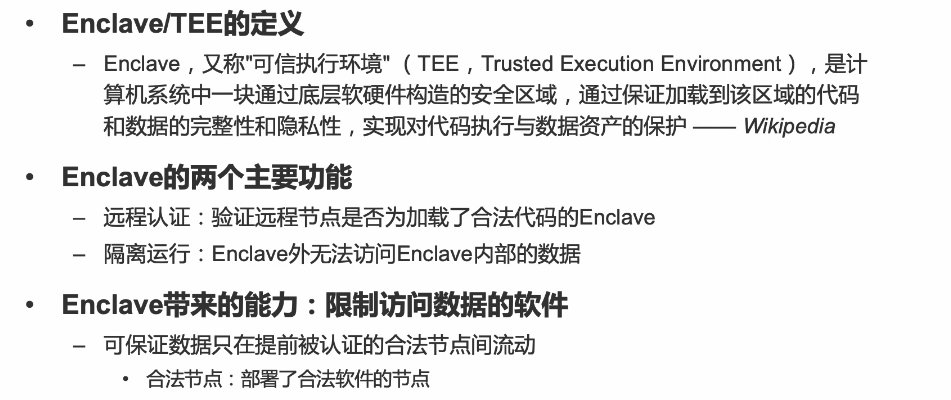


kexec：kernel的exec在kernel里面启一个kernel。

### 安全处理器



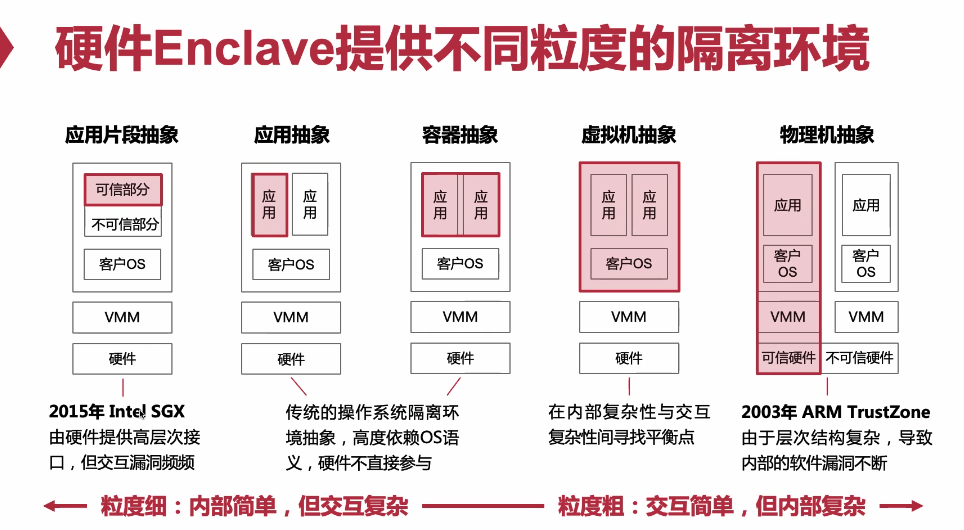
### TEE：可信执行环境



远程认证：要判断这个运行的地方是不是一个Enclave。需要一套硬件的机制来提供这个机制。在unclave里面埋一个key，做一个签名和公钥验证一下。（私钥不能出硬件）

隔离运行：里面运行的代码和数据不会被偷走。





Intel SGX：保护应用的user ISA。syscall这条指令不能在unclave里面运行。内外有个切换的过程，在OS运行，然后把结果放到一个内存里面去，然后unclave去读这块内存。在unclave里面运行的代码比较少。代码越少越安全：但你要和OS交互，你的access pattern就暴露了。

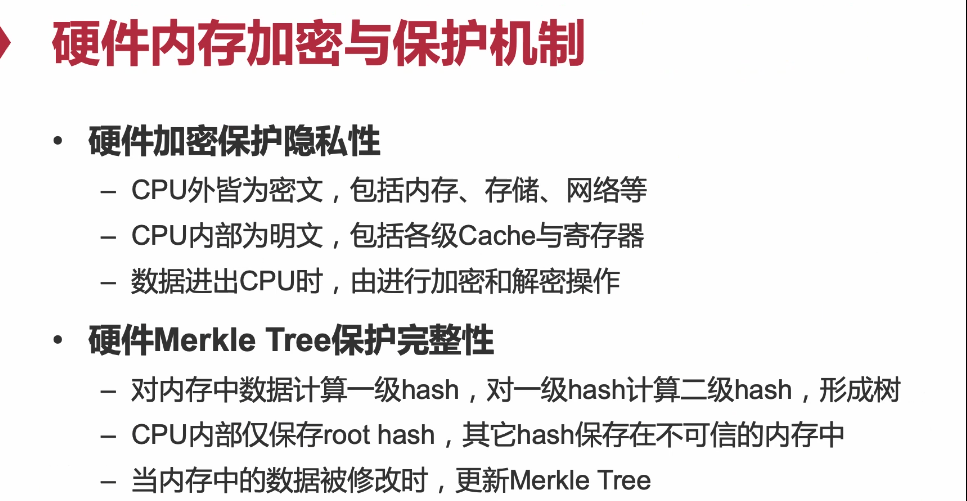
第二种应用（页表）或者容器（namespace）做隔离

虚拟机隔离：整个虚拟机就是一个实体，外面就是trap and emulate，语义是很低的，语义越低反推出用户数据就越难。

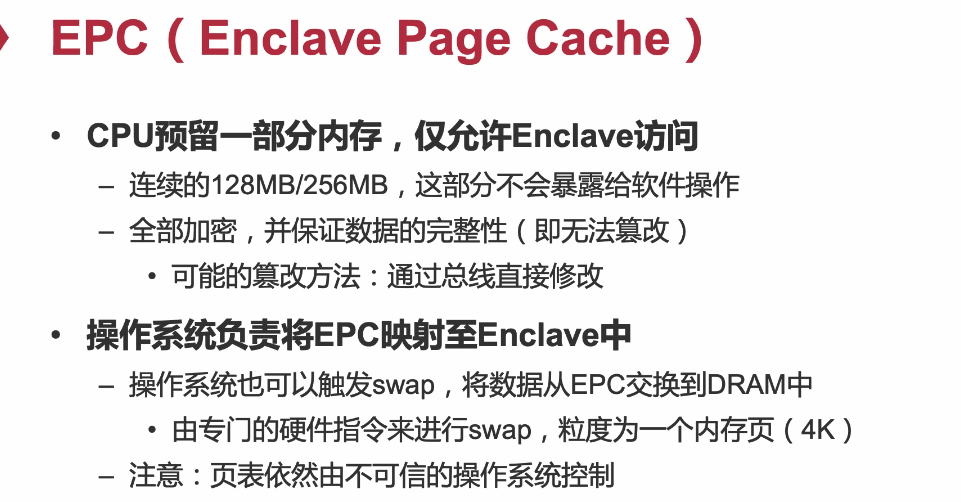
物理机抽象：左边右边可以跑完整的软件栈。

投屏：把显存切成两部分，投屏的时候不会把安全世界的显存投出去。

现在用的比较多的是虚拟机的抽象。可能再过几年，虚拟机变成unclave。



memory进cache做解密，解密CPU来做。在MMU旁边加一个加解密的引擎。然后为了防御攻击者随意篡改。对于SGX来说，CPU启动的时候预留内存，仅允许Enclave，OS不能访问。process reserved memory。



Enclave想换内存的时候，通过swap指令把。但copy出来的这个东西是加密的，有hash保护。依然有可能通过换顺序的方式，换出去的时候1，2，换回来的时候给你2，1。也有办法，用merkle tree，即使换了顺序，第一层hash对得上，第二层hash就对不上了。



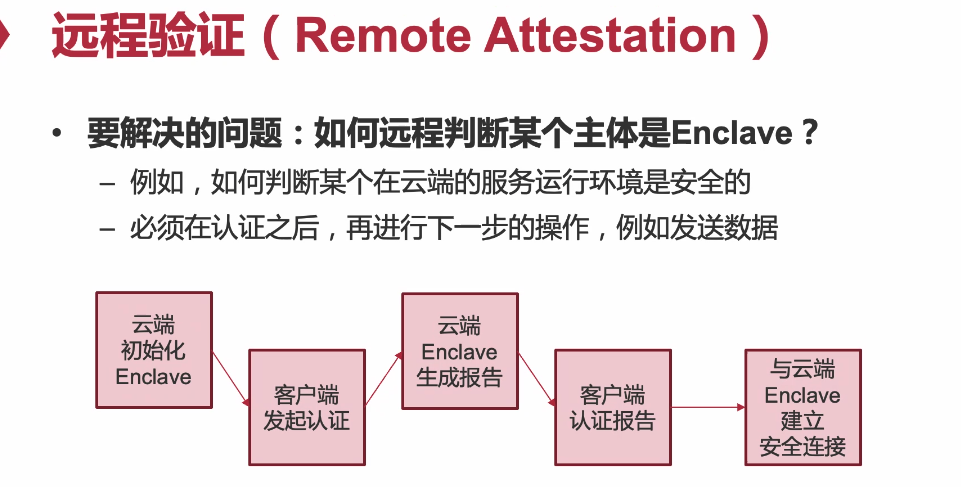
所有的执行都被限制在这一小块虚拟空间里。外面访问里面被CPU拒掉，里面访问外面可以。

Q：里面写外面不能写？

A：SFI。通过SFI提供一个软件的保障。

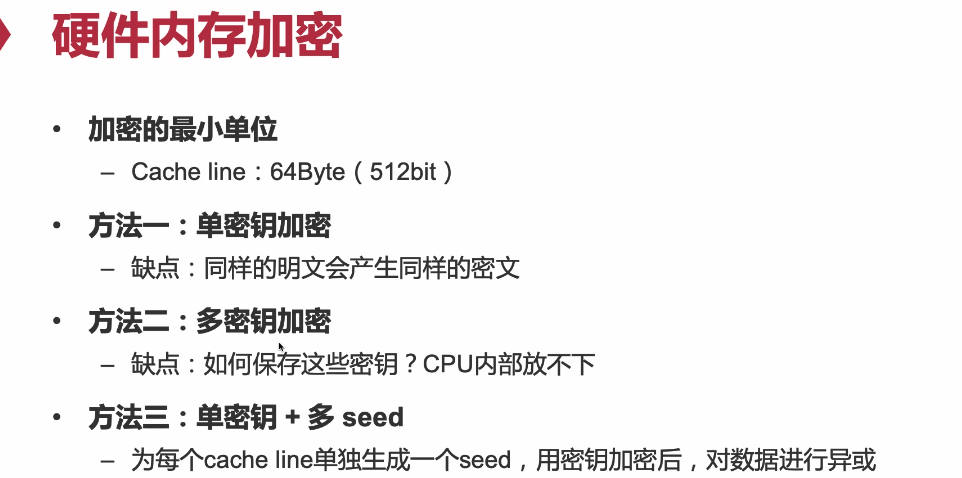
有了SGX之后，远程认证这个机制，你怎么知道远程和你交互的unclave是unclave？

你要发起一个认证，前提是阿里云已经把它的unclave初始化好了，阿里云拿到这个随机数之后，算一遍这个unclave里面的所有内存的hash。这是硬件提供的。

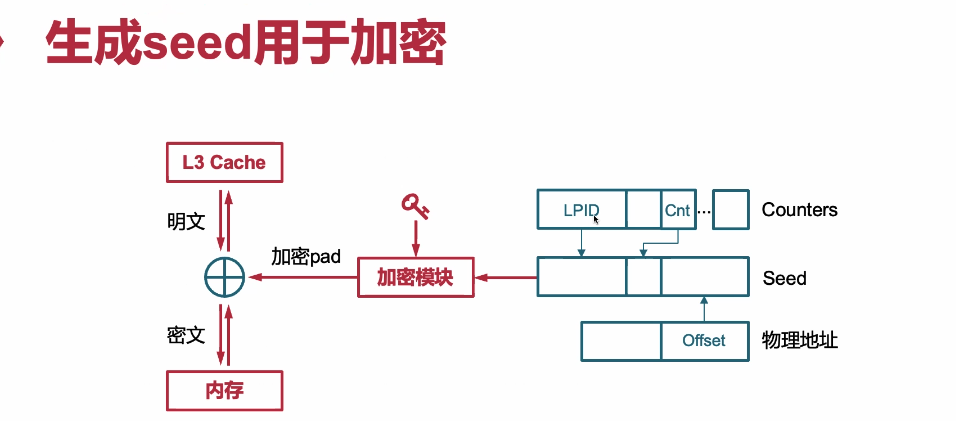


这个report去找intel，intel说我拿到一个report不知道这个私钥是真的还是假的，然后intel说你可以信。如果我跑在一个模拟器上面，有没有可能，但是跑在模拟器上面这个签名是假的签名。它是不可能是拿到一个真的private key的。除非有人把CPU内部的private key偷出来。不然它只是照着这个SGX的逻辑写了一个模拟器。这个key到底能不能偷出来呢？这就很tricky。

进内存加密不会很慢吗？

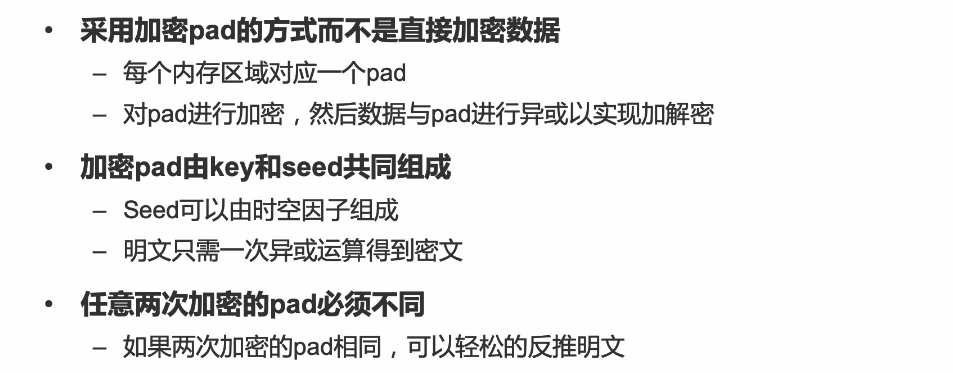


每一块内存都有不一样的秘钥。可以用秘钥加密seed，然后抑或这样产生的数据也是不一样的。



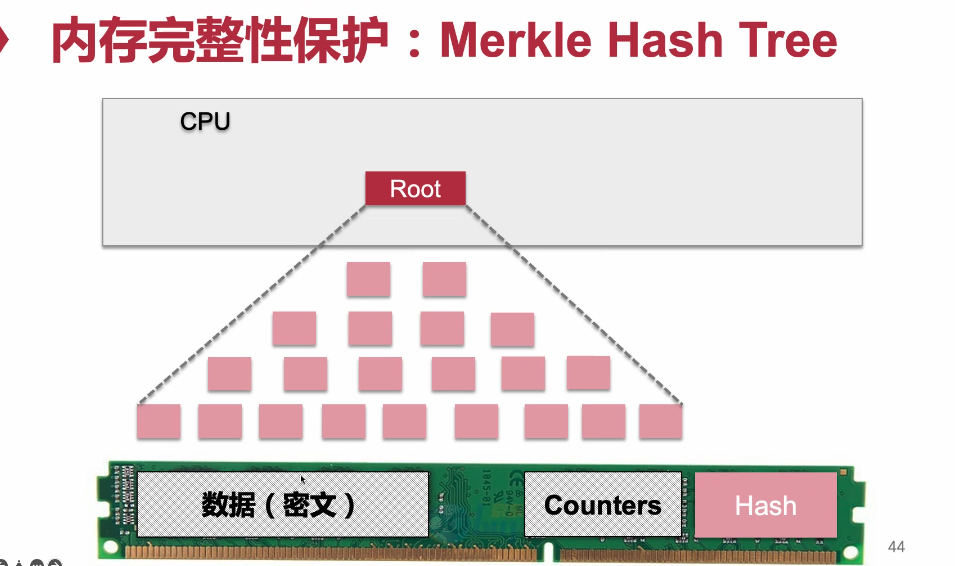
LPID：每一个页有一个ID。每一个cacheline加一个counter（会加一加一往上加）。Cnt大概7位，写一次内存就+1，每次写到cache也是不一样的。形成seed后用key加密，和内存的明文作抑或。只要做一次抑或就可以得到明文了，抑或是比较快的。cacheline访问内存的时候，访存本身需要不少cycle（cachemiss的时候才会加解密），这几百个cycle恰好可以用来做加解密——加密和访存可以做并发。然后访存完了，恰好做抑或。

如果LPID和Counter不在cache里，这就会很慢。LPID和Cnt高压缩比，每一个内存也对应64bytes。



seed和counter还有关系。





128M通过4k，4k算hash就能算出若干个hash。最终只有一个hash。最终会被保存在CPU里。读上来一个内存，会把它和其他的hash算一遍，看最终是不是得到那个root。可以把这几层hash都放到CPU里面，这样只要算相邻的hash就可以了。（放在CPU里面的root认为是可信的）。这也是为什么不是128M而不是128G，不然CPU里面放hash都要放很多东西。

SGX把hash去掉了，但是可以支持64G的memory。