中山大學 软件工程学院
SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

操作系统安全

SSE202/204: 操作系统原理

苏玉鑫
suyx35@mail.sysu.edu.cn
助教：龙玉丹 单诗雯 毛晨希 沈志轩 郑灿峰 胡伟峰

版权信息

- 部分内容来自：上海交通大学并行与分布式系统研究所操作系统课件
  - https://ipads.se.sjtu.edu.cn/courses/os/
- 其它参考资料：
  - 清华大学操作系统公开课
    - https://open.163.com/newview/movie/courseintro?newurl=ME1NSA351
    - 介绍标准内容，适合考研
  - 南京大学计算机软件研究所
    - http://jyywiki.cn/OS/2025/
    - https://space.bilibili.com/202224425/channel/collectiondetail?sid=192498
    - 比较有趣

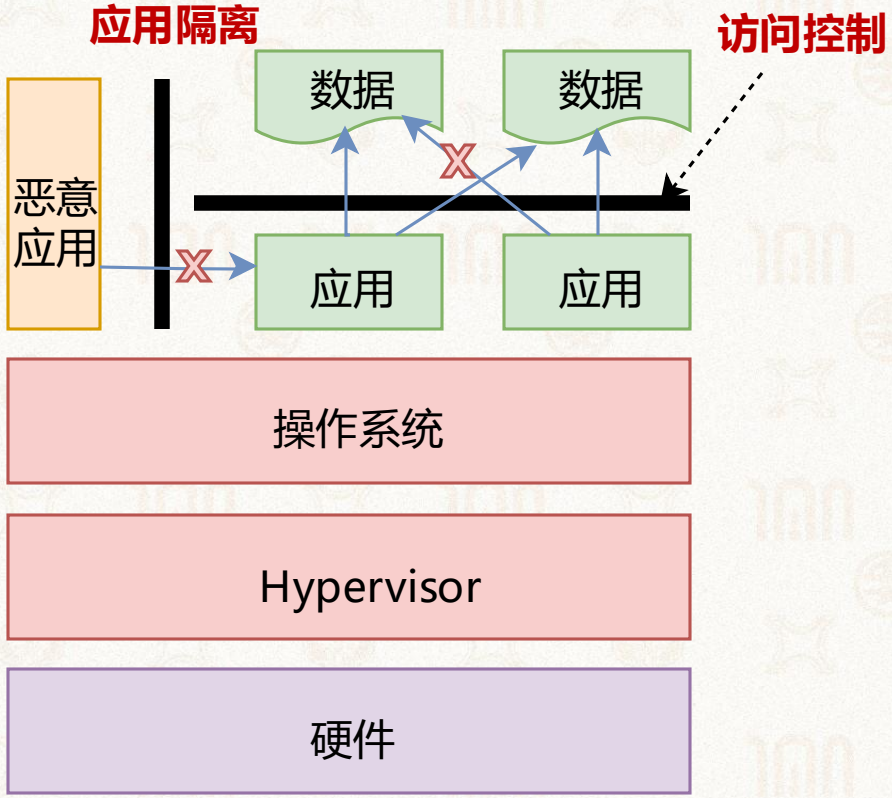
大纲

- 基本概念
  - 操作系统的安全层次
  - 访问控制模型
  - SELinux
- 操作系统内部安全
  - 系统漏洞
  - 侧信道、隐秘信道
  - 缓存信道攻击
  - 缓存信道防御
  - Meltdown经典案例
- 恶意操作系统

安全是操作系统的重要功能和服务

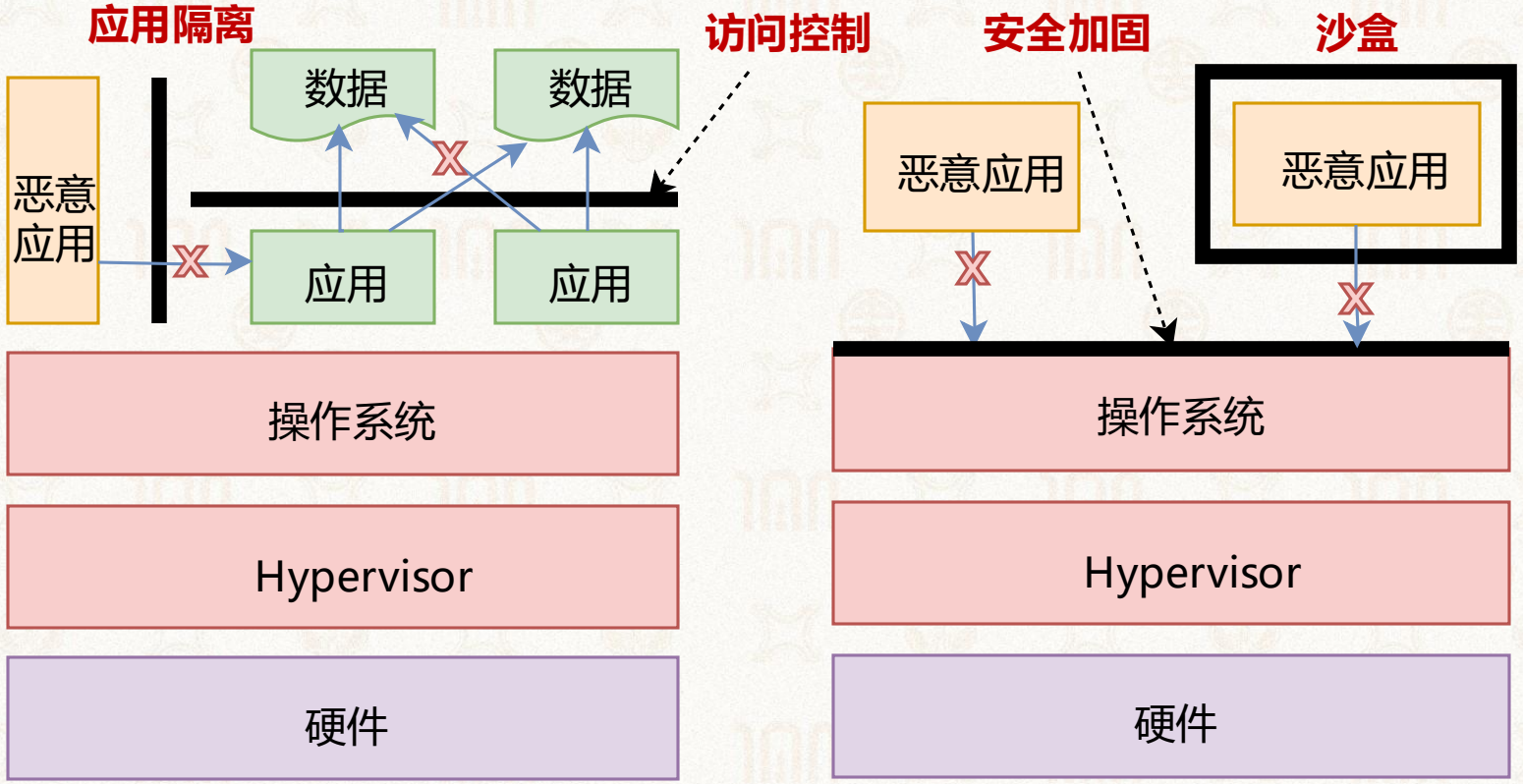
- 系统中有许多需要保护的数据
  - 如账号密码、信用卡号、地理位置、照片视频等
  - 操作系统需要允许这些数据被合法访问，但不允许被非法访问
- 系统中可能存在许多恶意应用
  - 操作系统需要与这些恶意应用作斗争，保护自己，限制对方
- 操作系统不可避免的存在漏洞
  - 操作系统需要考虑自己被完全攻破的情况下依然提供一定的保护

操作系统安全的三个层次



基于操作系统的应用隔离与访问控制

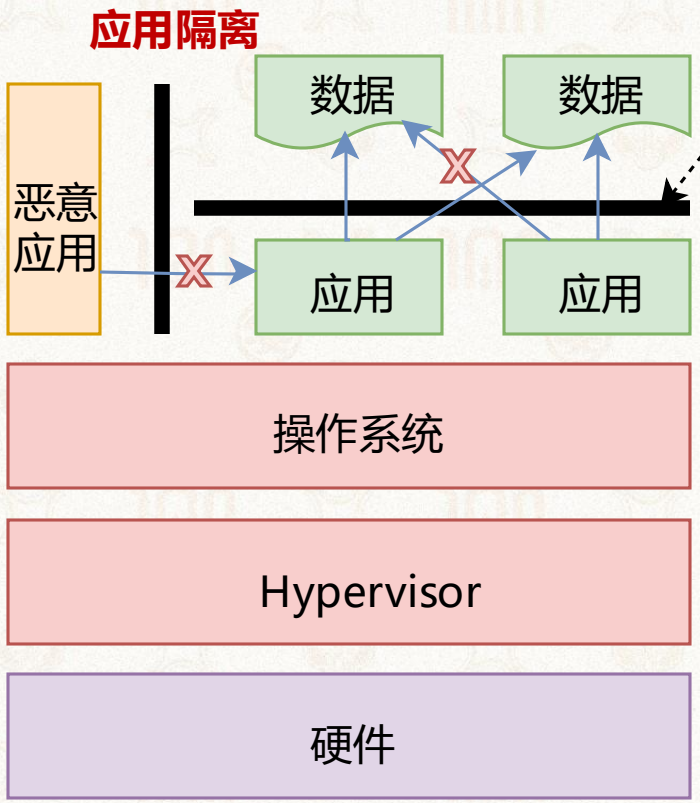
操作系统安全的三个层次



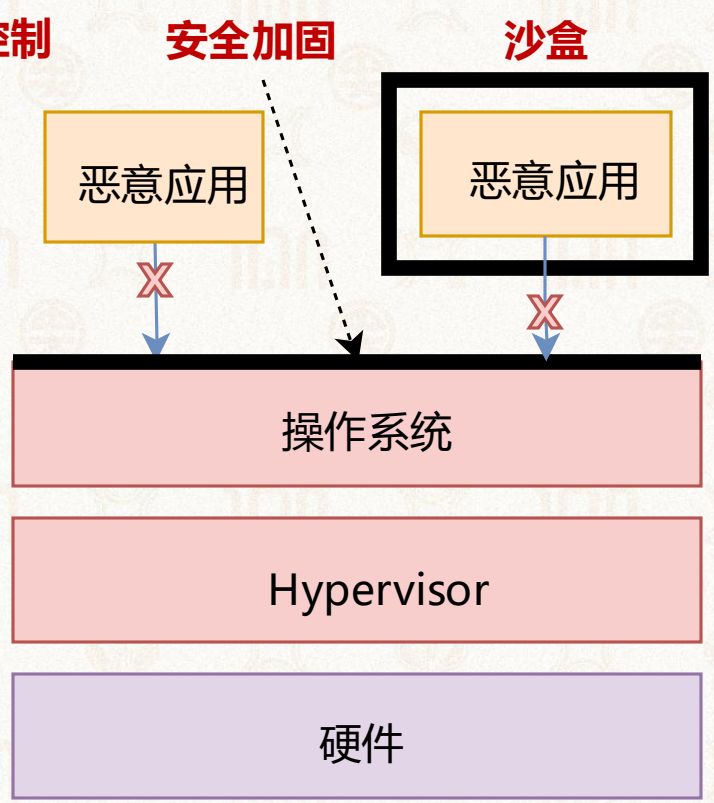
基于操作系统的应用隔离与访问控制

操作系统对恶意应用的隔离与防御

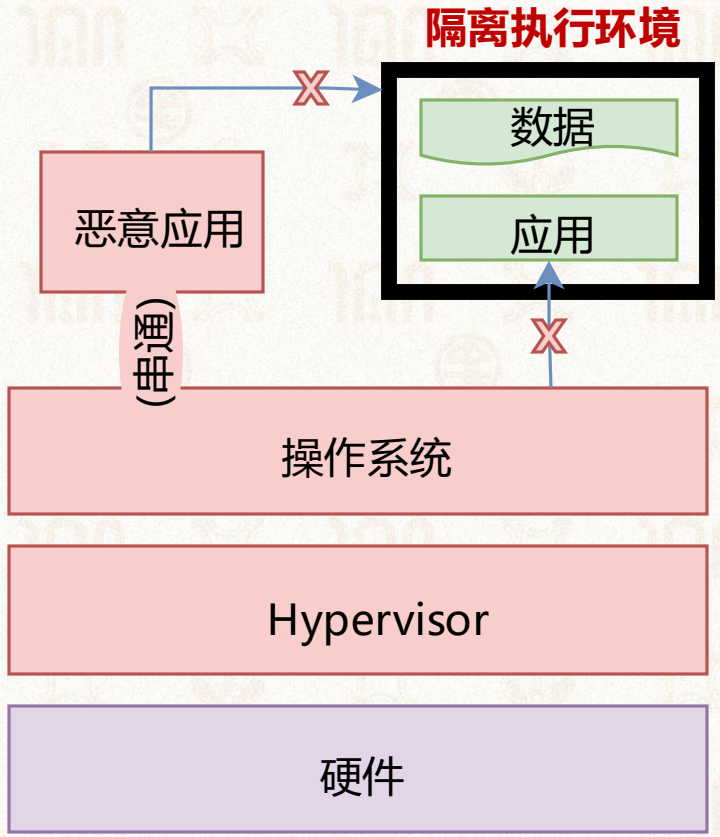
操作系统安全的三个层次



基于操作系统的应用隔离与访问控制



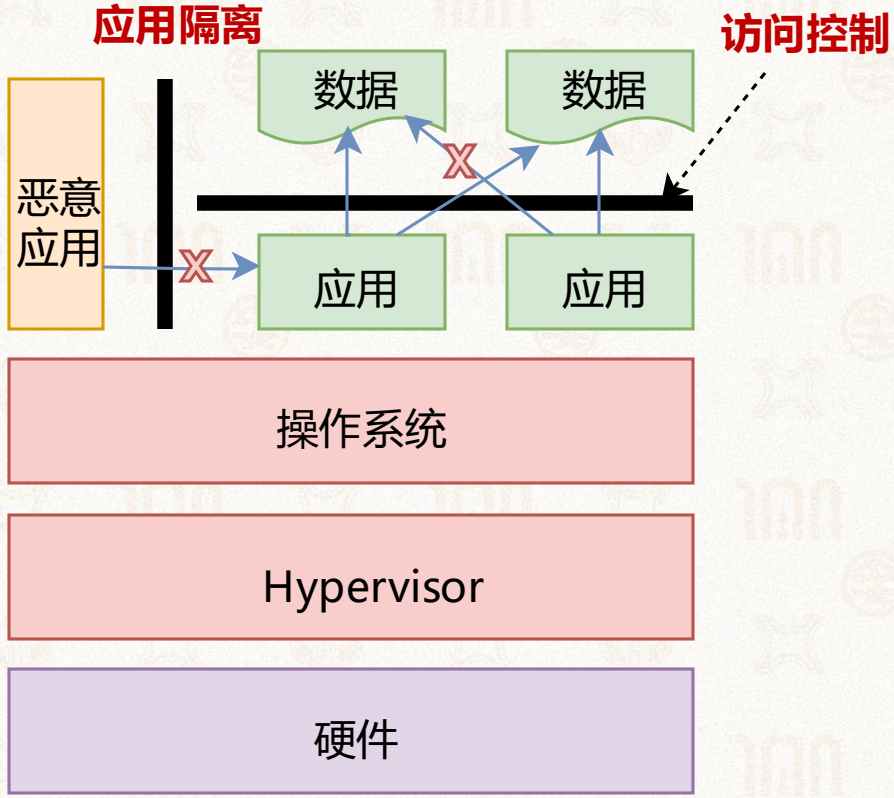
操作系统对恶意应用的隔离与防御



操作系统不可信时对应用的保护

层次一：基于OS的应用隔离与访问控制

- 威胁模型
  - 操作系统是可信的，能够正常执行且不受攻击
  - 应用程序可能是恶意的，会窃取其他应用数据
  - 应用程序可能存在bug，导致访问其他应用数据

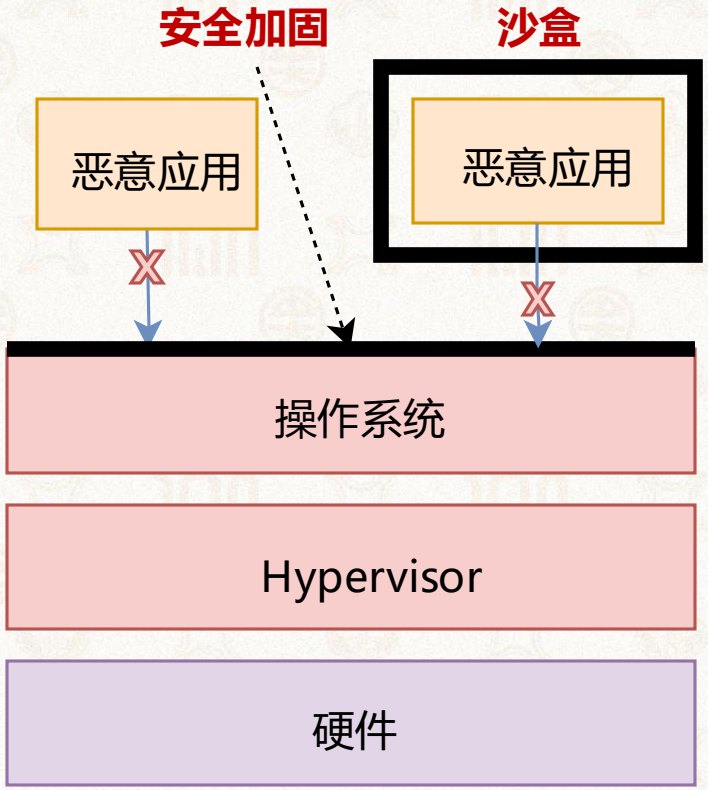


- 应用隔离
  - 内存数据隔离：依赖进程间不同虚拟地址空间的隔离
  - 文件系统隔离：文件系统是全局的，需限制哪些应用不能访问哪些文件
    - 操作系统提供对文件系统的访问控制机制

层次二：OS对恶意应用的隔离与防御

- 威胁模型
  - 操作系统存在bug和安全漏洞
  - 操作系统的运行过程依然可信
  - 恶意应用利用操作系统漏洞攻击，获取更高权限或直接窃取其他应用的数据

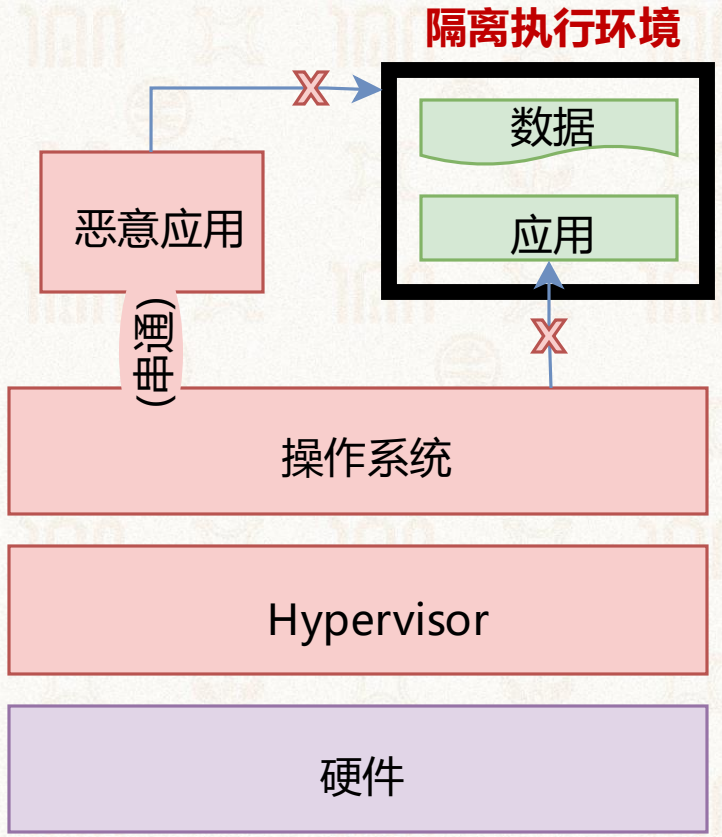
- 操作系统防御
  - 防御常见的操作系统bug/漏洞
  - 沙盒机制限制应用的运行



层次三：OS不可信时对应用的保护

- 威胁模型
  - 操作系统不可信，有可能被攻击者完全控制
  - 恶意应用可能与操作系统串通发起攻击

- 基于更底层的应用保护
  - 基于Hypervisor的保护：可信基更小
  - 基于硬件Enclave的保护：硬件通常更可信



操作系统安全的三个概念

- 可信计算基（Trusted Computing Base）
  - 为实现计算机系统安全保护的所有安全保护机制的集合
  - 包括软件、硬件和固件（硬件上的软件）

- 攻击面（Attacking Surface）
  - 一个组件被其他组件攻击的所有方法的集合
  - 可能来自上层、同层和底层

- 防御纵深（Defense in-depth）
  - 为系统设置多道防线，为防御增加冗余，以进一步提高攻击难度

大纲

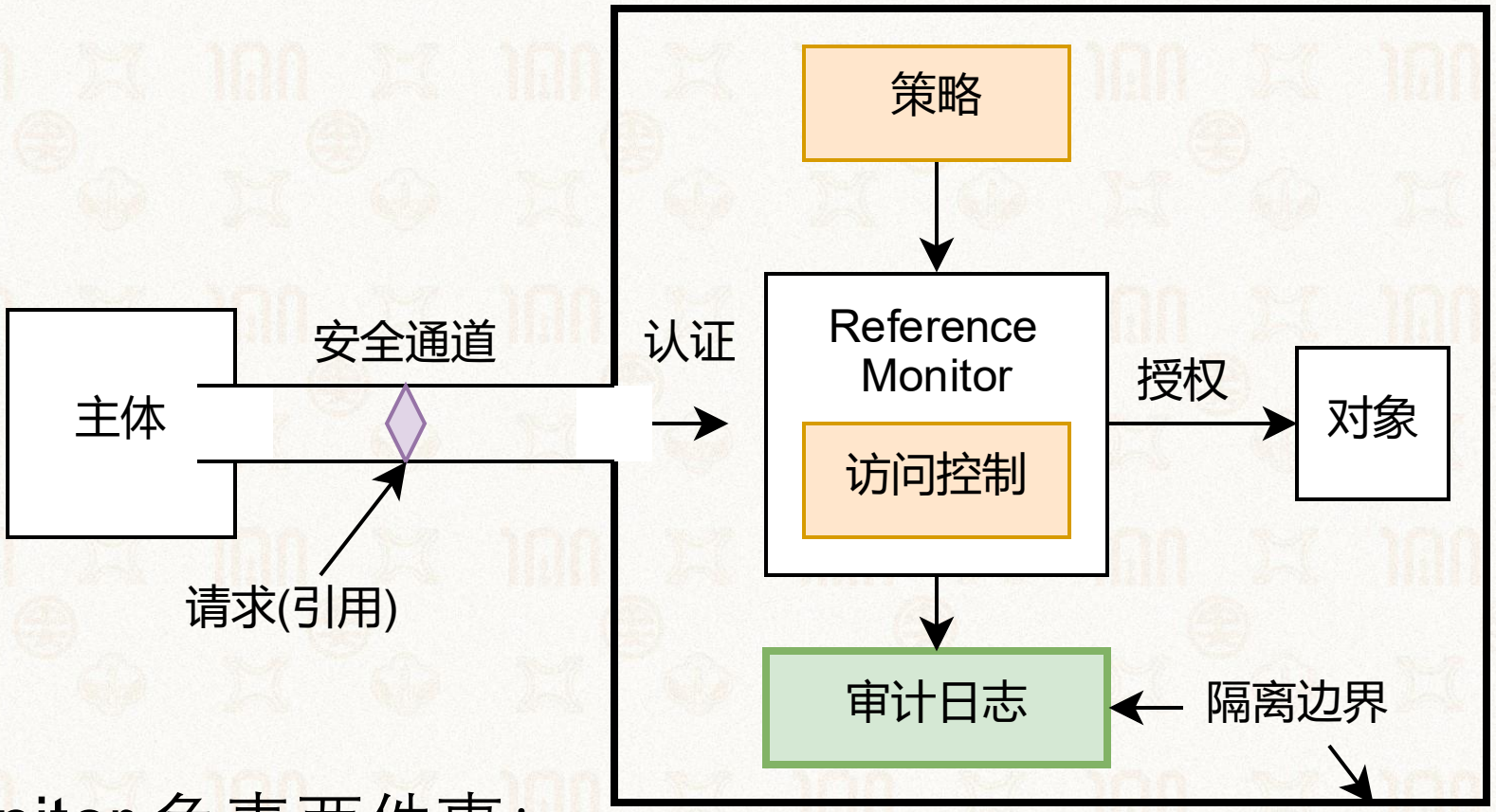
- 基本概念
  - 操作系统的安全层次
  - 访问控制模型
  - SELinux
- 操作系统内部安全
  - 系统漏洞
  - 侧信道、隐秘信道
  - 缓存信道攻击
  - 缓存信道防御
  - Meltdown经典案例
- 恶意操作系统

访问控制与引用监视器

- 访问控制（Access Control）
  - 按照访问实体的身份来限制其访问对象的一种机制
  - 为了实现对不同应用访问不同数据的权限控制
  - 包含"认证"和"授权"两个重要步骤

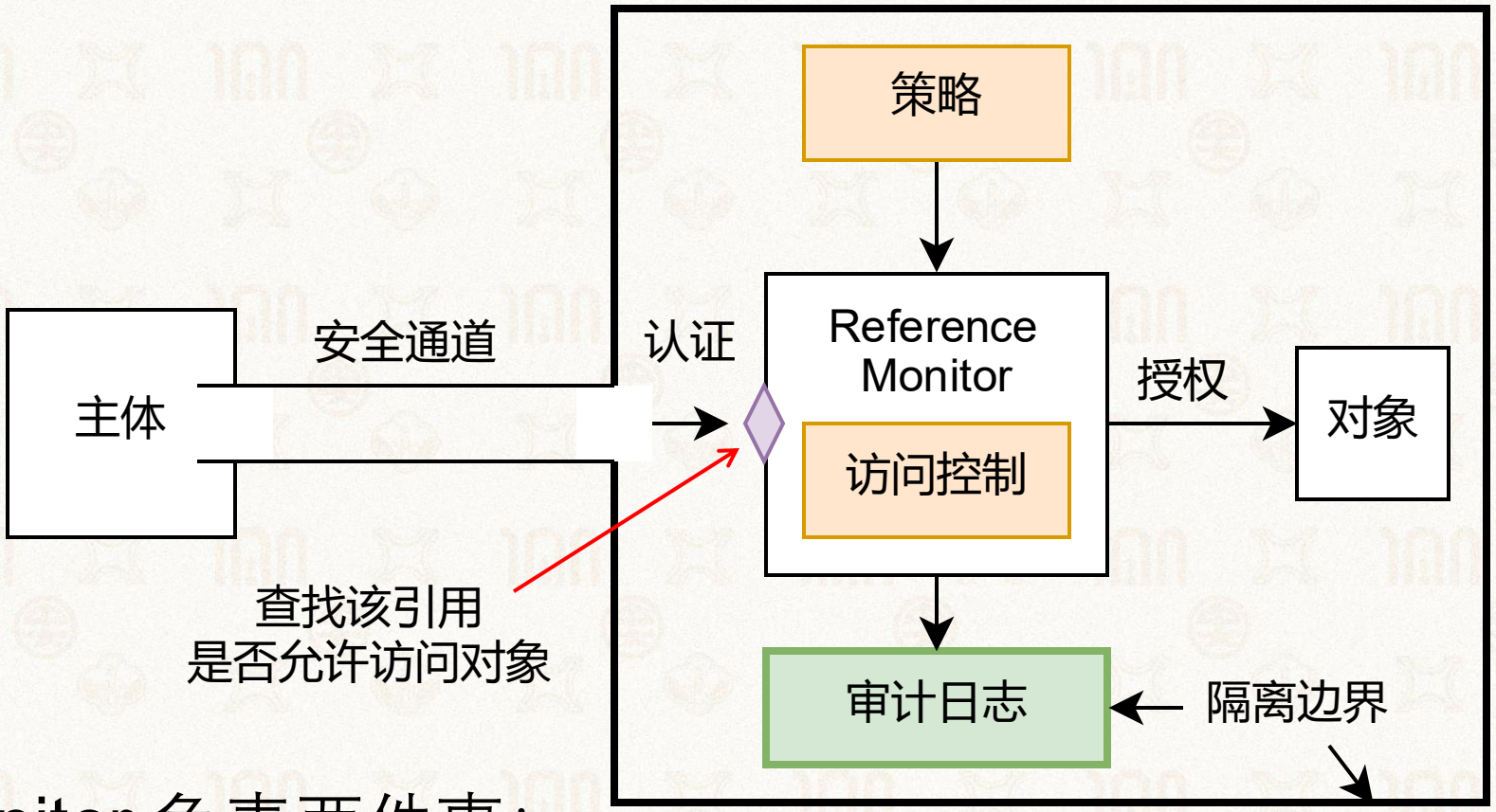
- 引用监视器（Reference Monitor）
  - 是实现访问控制的一种方式
  - 主体必须通过引用（reference）的方式间接访问对象
  - Reference monitor 位于主体和对象之间，进行检查

引用监视器（Reference Monitor）机制



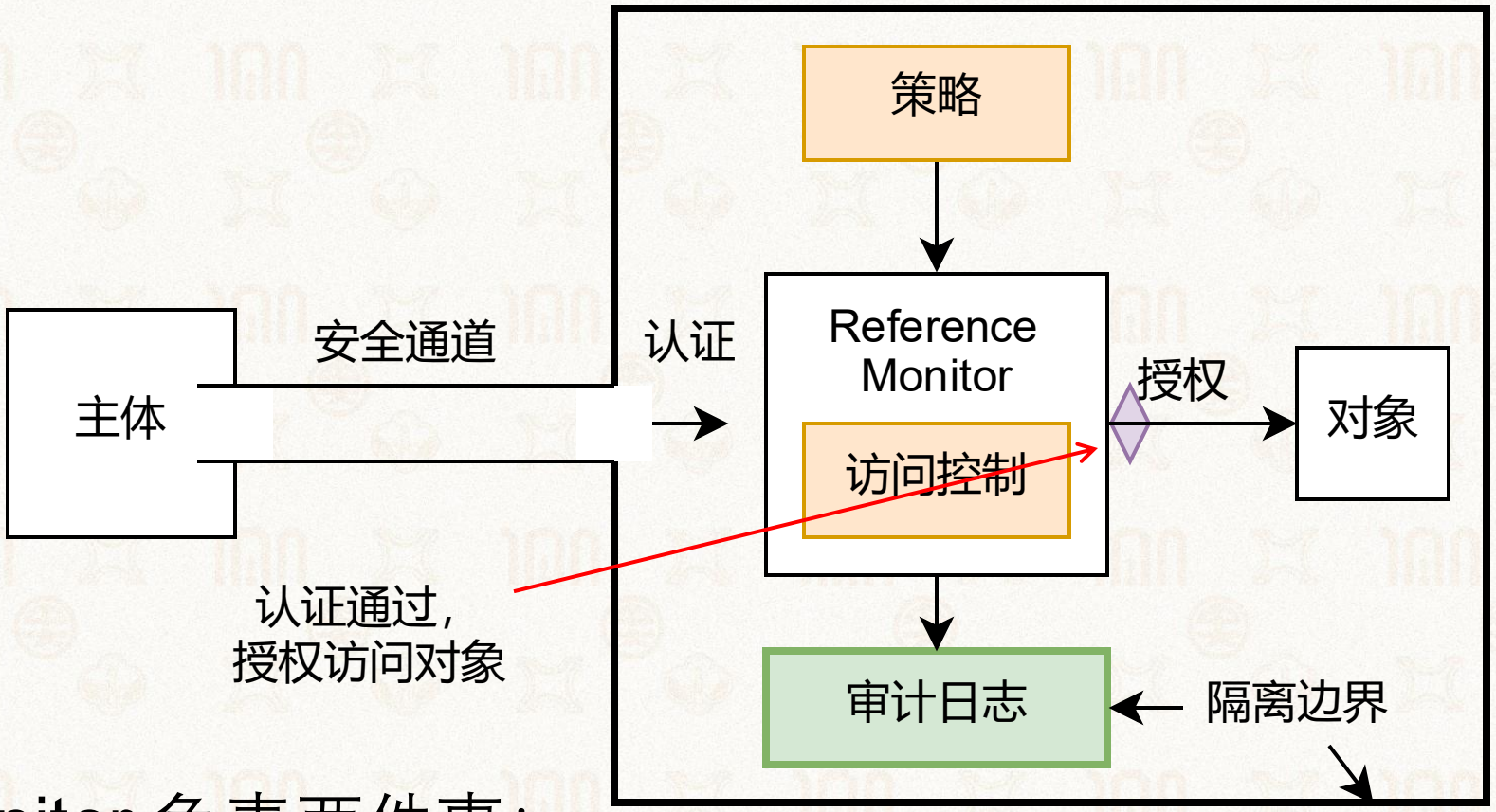
- Reference Monitor 负责两件事：
  - Authentication：确定发起请求实体的身份，即认证
  - Authorization：确定实体确实拥有访问资源的权限，包含授权和鉴权

引用监视器（Reference Monitor）机制



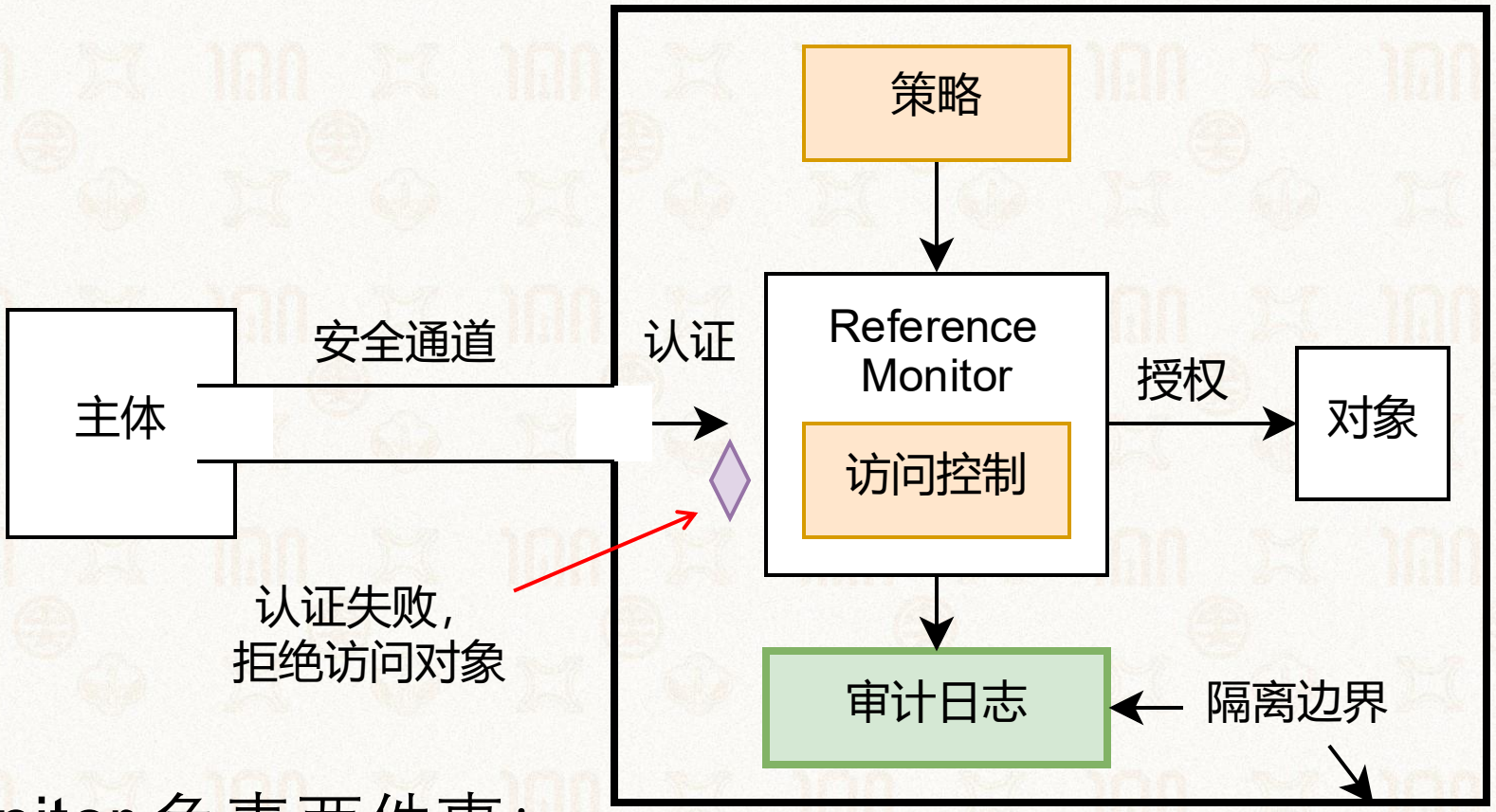
- Reference Monitor 负责两件事：
  - Authentication：确定发起请求实体的身份，即认证
  - Authorization：确定实体确实拥有访问资源的权限，包含授权和鉴权

引用监视器（Reference Monitor）机制



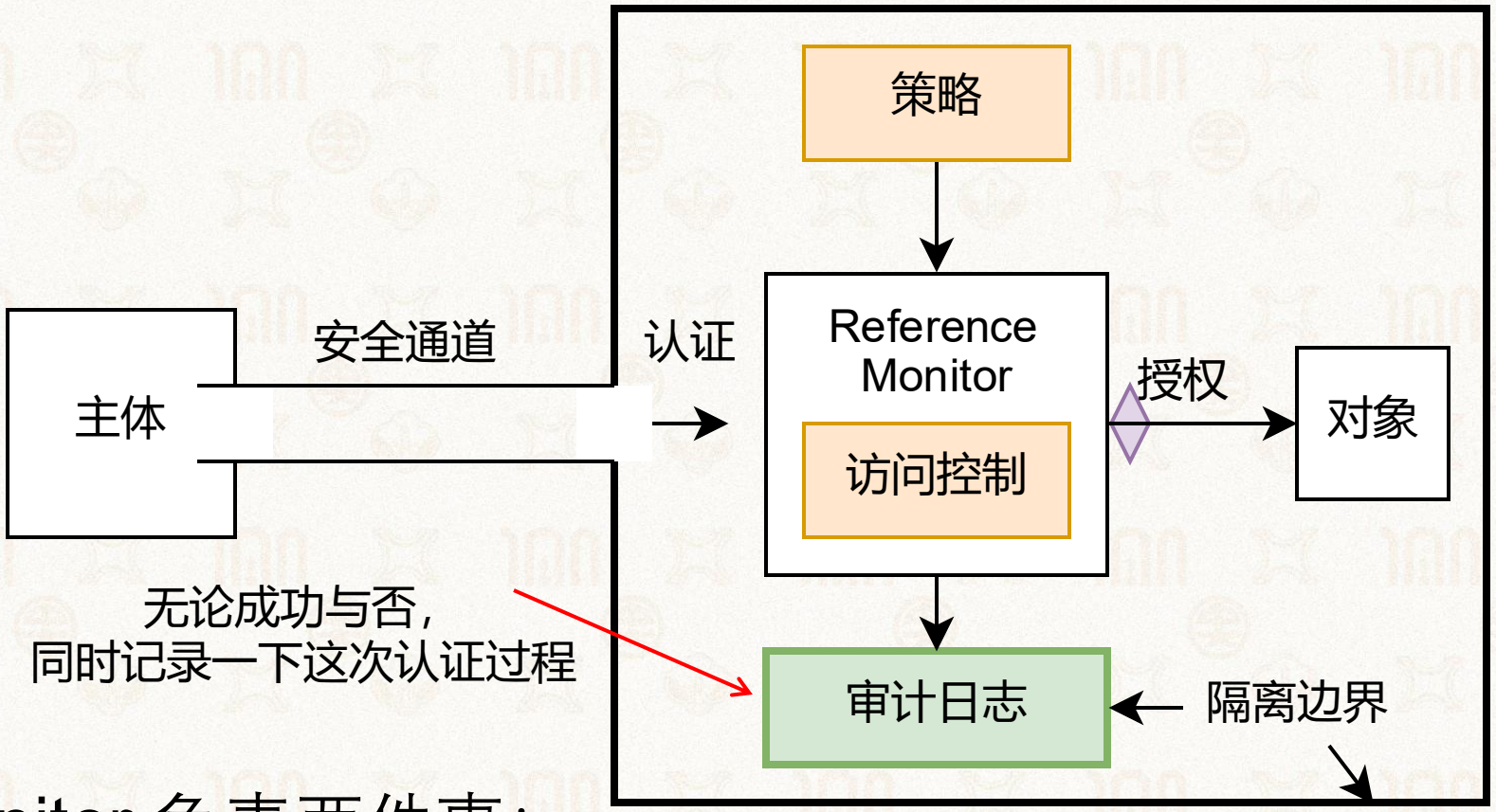
- Reference Monitor 负责两件事：
  - Authentication：确定发起请求实体的身份，即认证
  - Authorization：确定实体确实拥有访问资源的权限，包含授权和鉴权

引用监视器（Reference Monitor）机制



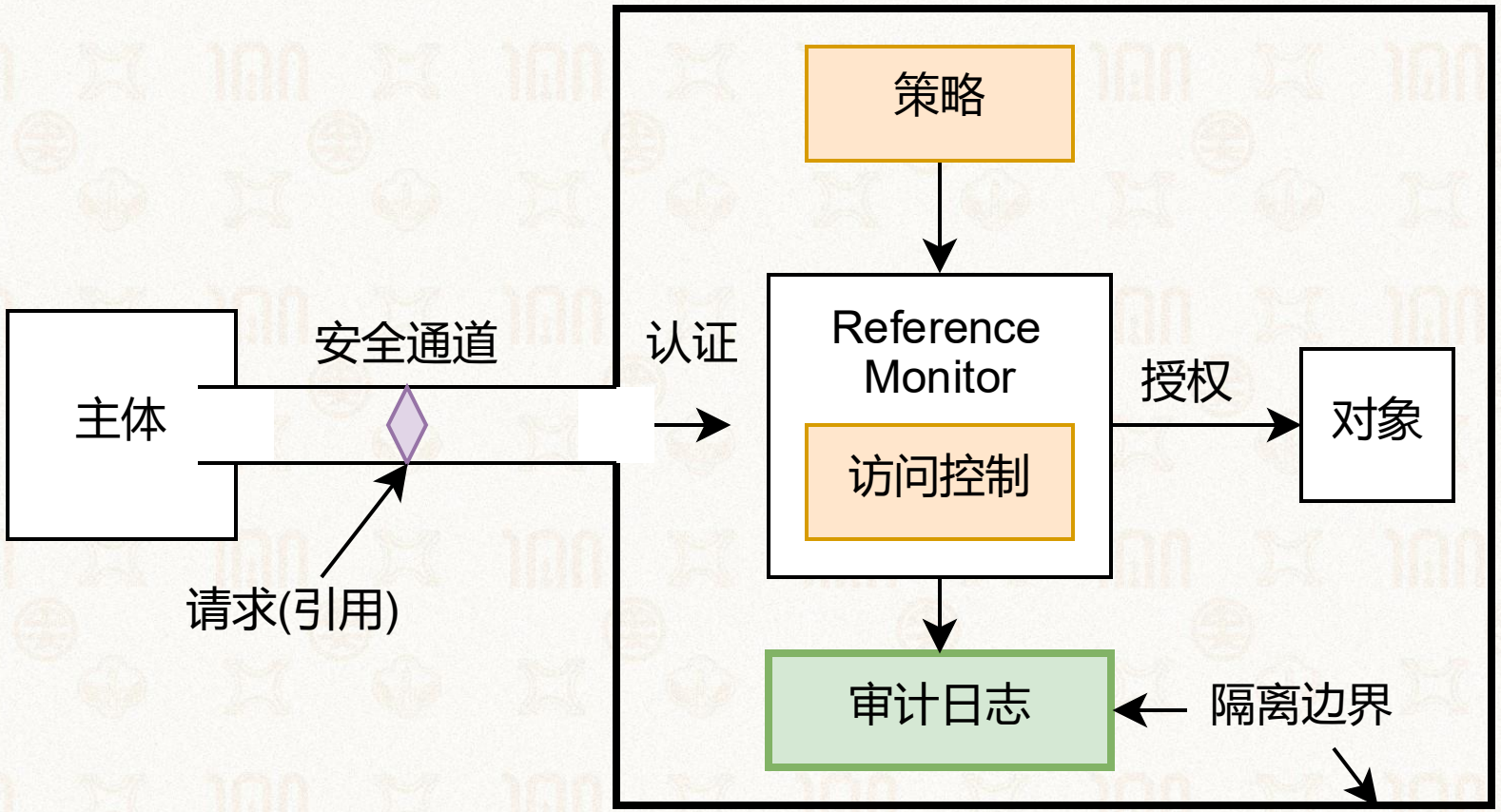
- Reference Monitor 负责两件事：
  - Authentication：确定发起请求实体的身份，即认证
  - Authorization：确定实体确实拥有访问资源的权限，包含授权和鉴权

引用监视器（Reference Monitor）机制



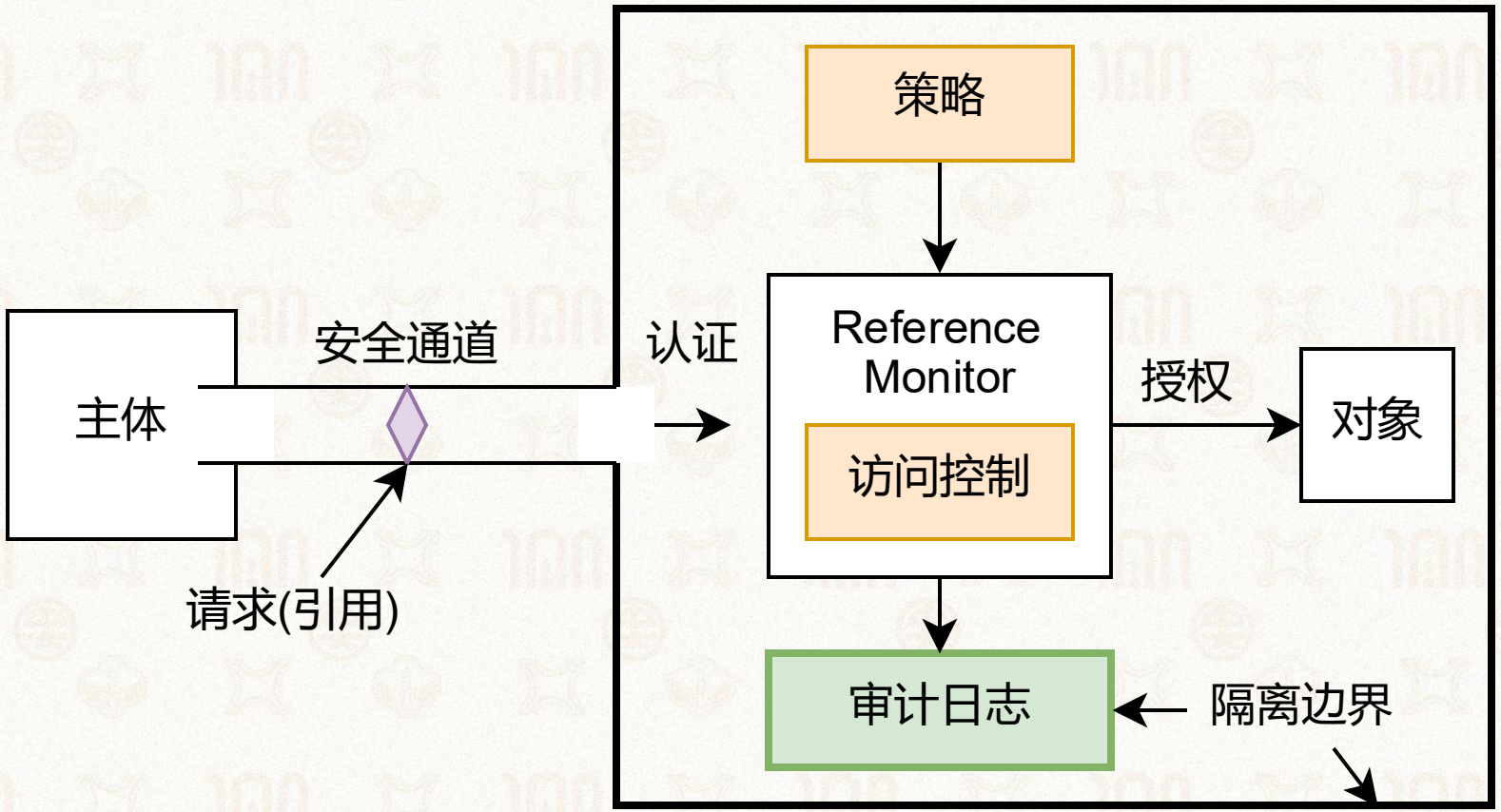
- Reference Monitor 负责两件事：
  - Authentication：确定发起请求实体的身份，即认证
  - Authorization：确定实体确实拥有访问资源的权限，包含授权和鉴权

引用监视器（Reference Monitor）机制



- 考虑一个问题：文件为什么用“路径”表示，而不是由inode编号表示？

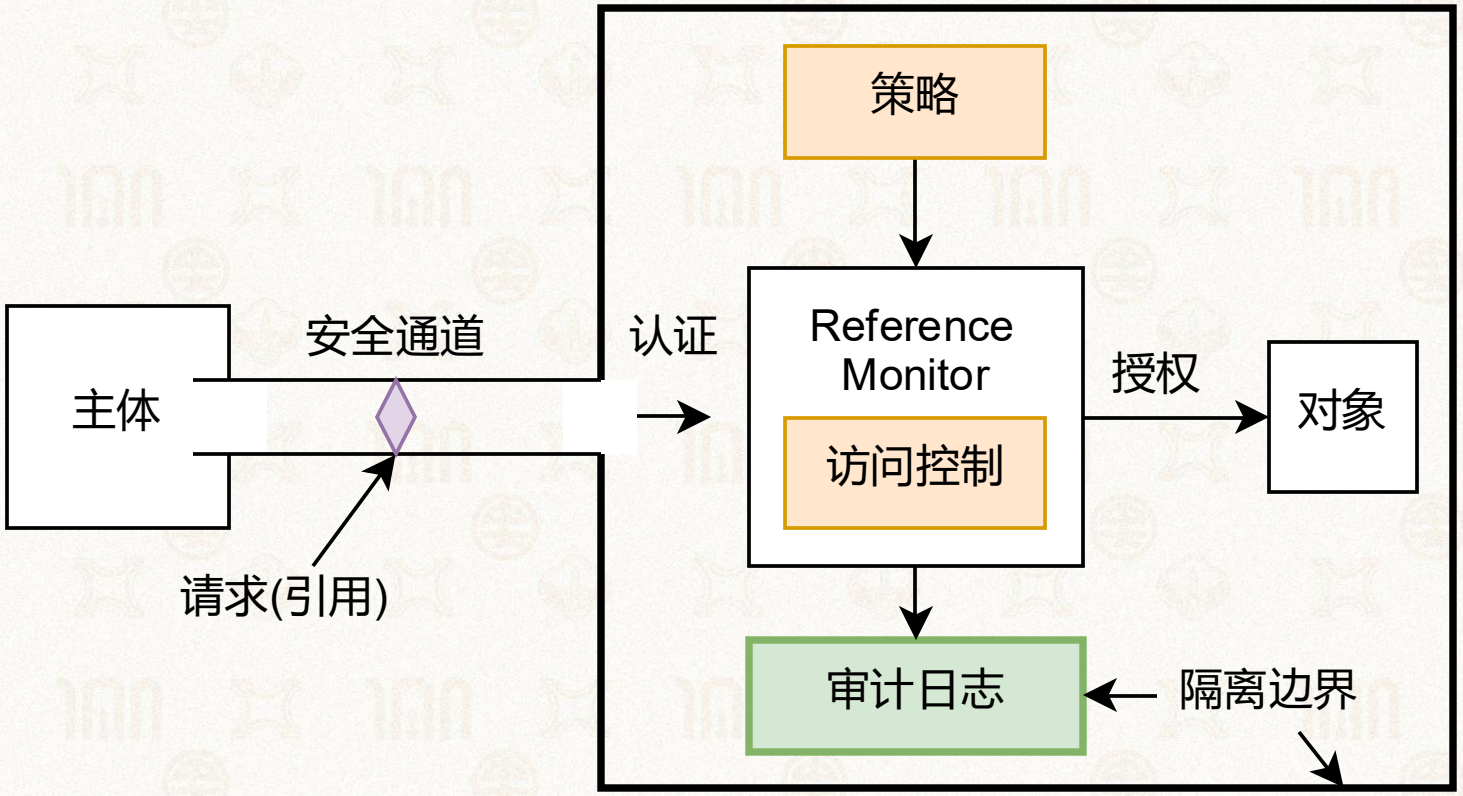
引用监视器（Reference Monitor）机制



- 考虑一个问题：文件为什么用“路径”表示，而不是由inode编号表示？
- 答：“路径”就是一种引用，可以用来做权限管理

认证机制

- 知道什么
  - 例如密码/口令、手势密码、某个问题的答案等
  - 密码容易泄露
- 有什么
  - 例如 USB-key、密码器等实物
- 是什么
  - 如指纹、虹膜、步态、键盘输入习惯等属于人的一部分



授权机制

- 权限矩阵
  - 对象与实体的关系

文件

	对象-1	对象-1	对象-3
实体-1	读/写	读/执行	读
实体-2		读/执行	读/写
实体-3	读		读/写

比如用户、进程

- 矩阵有多大？
  - 假如系统中有 100 个用户，每种权限用 1 个 bit 来表示，那么每个文件都 至少需要 300 个 bit 来表示 100 个用户 的 3 种权限
  - 假设这些 bit 都保存在 inode 中，通常 inode 的大小为 128-Byte 或 256-Byte，300 个 bit 相当于一个 inode 的 15% 至 30%
  - 每当新建一个用户的时候，都必须要更新所有 inode 中的权限 bit，不现实

授权机制：以文件系统为例

- 使用"用户组"的概念，将用户分为三类
  - 文件拥有者、文件拥有组、其他用户组
  - 每个文件只需要用9个bit即可：3种权限（读-写-执行） x 3 类用户

- 何时检查用户权限？
  - 每次打开文件时，进行鉴权和授权
    - open()包含可读/可写的参数，OS根据用户组进行检查（鉴权）
    - 引入fd，记录本次打开权限（授权），作为后续操作的参数
  - 每次操作文件时，根据fd信息进行检查（鉴权）
  - 所以fd就是一个引用

```
int open(const char* path, int oflag, ...);
```

```
yxsu@Dell-T6401:~/os/process$ ls -al
总用量 192
drwxrwxr-x 2 yxsu yxsu  4096 4月  27 02:27 .
drwxrwxr-x 4 yxsu yxsu  4096 4月  24 21:27 ..
-rwxrwxr-x 1 yxsu yxsu 16920 3月  23 02:27 a.out
-rwxrwxr-x 1 yxsu yxsu 16792 3月  21 02:28 execve_demo
-rw-rw-r-- 1 yxsu yxsu   431 3月  21 02:28 execve_demo.c
-rwxrwxr-x 1 yxsu yxsu 16832 3月  21 17:38 fork_demo
-rw-rw-r-- 1 yxsu yxsu   456 3月  21 00:10 fork_demo.c
-rwxrwxr-x 1 yxsu yxsu 16904 3月  21 17:47 fork_readfile
-rw-rw-r-- 1 yxsu yxsu   443 3月  21 00:42 fork_readfile.c
-rwxrwxr-x 1 yxsu yxsu 16840 3月  20 21:44 hello-name
-rw-rw-r-- 1 yxsu yxsu   164 3月  20 21:40 hello-name.c
-rwxrwxr-x 1 yxsu yxsu 16744 3月  21 02:30 myecho
-rw-rw-r-- 1 yxsu yxsu   205 3月  21 02:30 myecho.c
```

最小特权级原则：SUID 机制

- 问题：passwd 命令如何工作？
  - 用户有权限使用 passwd 命令修改自己的密码
  - 用户的密码保存在 /etc/shadow 中，用户无权访问
  - 本质上是以文件为单位的权限管理粒度过粗——怎么解决？

- 解决方法：运行 passwd 时使用 root 身份
  - 如何保证用户提权为root后只能运行passwd？
    - 在passwd的inode中增加一个SUID位，使得用户仅在执行该程序时才会被提权，从而将进程提权的时间降至最小
  - passwd程序本身的逻辑会保证某一个用户只能修改其自身的密码

```
yxsu@Dell-T6401:~$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 68208 7月  15  2021 /usr/bin/passwd
```

SUID位

基于角色的访问控制（RBAC）

- RBAC：将用户与角色解耦的访问控制方法
  - 提出了角色的概念，与权限直接相关
  - 用户通过拥有一个或多个角色间接地拥有权限
  - "用户-角色"，以及"角色-权限"，一般都是多对多的关系
  - 例如，“管理员”角色的权限肯定比“普通用户”角色要高

- RBAC的优势
  - 设定角色与权限之间的关系比设定用户与权限之间的关系更直观
  - 可一次性地更新所有拥有该角色用户的权限，提高了权限更新的效率
  - 角色与权限之间的关系比较稳定，而用户和角色之间的关系变化相对频繁
    - 设计者负责设定权限与角色的关系（机制）
    - 管理者只需要配置用户属于哪些角色（策略）

DAC与MAC

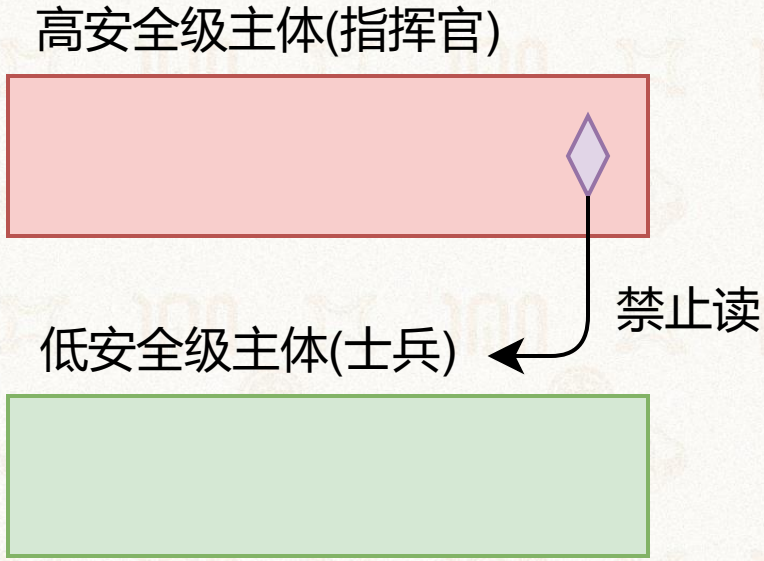
- 自主访问控制（DAC: Discretionary Access Control）
  - 指一个对象的拥有者有权限决定该对象是否可以被其他人访问
    - 例如，文件系统就是一类典型的 DAC
  - 对部分场景（如军队）来说过于灵活
    - 例如，文件拥有者可随意设置文件权限

- 强制访问控制（MAC: Mandatory Access Control）
  - 由系统增加一些强制的、不可改变的规则
    - 例如，在军队中，如果某个文件设置为机密，那么就算是指挥官也不能把这个文件给没有权限的人看——这个规则是由军法（系统）规定的
  - MAC与DAC可以结合，MAC的优先级更高

Bell-LaPadula 模型

- BLP属于强制访问控制（MAC）模型
  - 一个用于访问控制的状态机模型
  - 目的是为了用于政府、军队等具有严格安全等级的场景

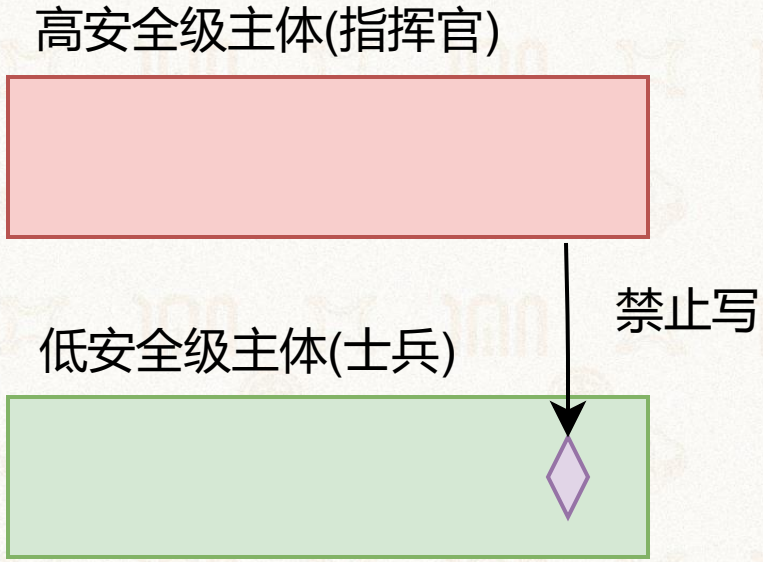
- BLP 规定了两条 MAC 规则和一条 DAC 规则：
  - 简单安全属性：某个安全级别的主体无法读取更高安全级别的对象
  - * 属性（星属性）:某一安全级别的主体无法写入任何更低安全级别的对象
  - 自主安全属性：使用访问矩阵来规定自主访问控制（DAC）



Bell-LaPadula 模型

- BLP属于强制访问控制（MAC）模型
  - 一个用于访问控制的状态机模型
  - 目的是为了用于政府、军队等具有严格安全等级的场景

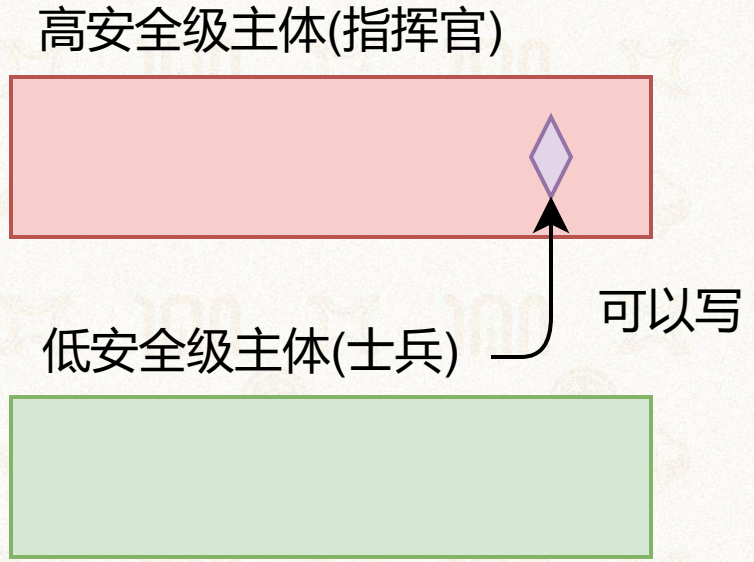
- BLP 规定了两条 MAC 规则和一条 DAC 规则：
  - 简单安全属性：某个安全级别的主体无法读取更高安全级别的对象
  - * 属性（星属性）:某一安全级别的主体无法写入任何更低安全级别的对象
  - 自主安全属性：使用访问矩阵来规定自主访问控制（DAC）



Bell-LaPadula 模型

- BLP属于强制访问控制（MAC）模型
  - 一个用于访问控制的状态机模型
  - 目的是为了用于政府、军队等具有严格安全等级的场景

- BLP 规定了两条 MAC 规则和一条 DAC 规则：
  - 简单安全属性：某个安全级别的主体无法读取更高安全级别的对象
  - * 属性（星属性）:某一安全级别的主体无法写入任何更低安全级别的对象
  - 自主安全属性：使用访问矩阵来规定自主访问控制（DAC）

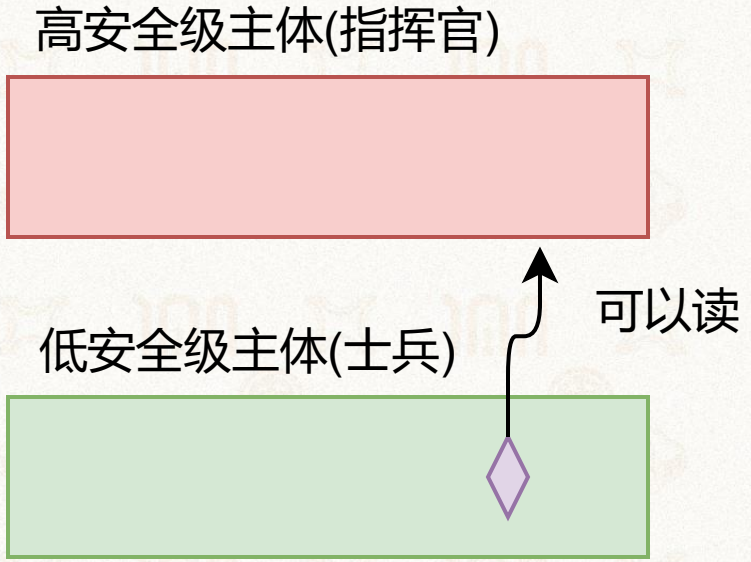


注意箭头：信息流可以从低级到高级，但高级的信息不能流出

Bell-LaPadula 模型

- BLP属于强制访问控制（MAC）模型
  - 一个用于访问控制的状态机模型
  - 目的是为了用于政府、军队等具有严格安全等级的场景

- BLP 规定了两条 MAC 规则和一条 DAC 规则：
  - 简单安全属性：某个安全级别的主体无法读取更高安全级别的对象
  - * 属性（星属性）:某一安全级别的主体无法写入任何更低安全级别的对象
  - 自主安全属性：使用访问矩阵来规定自主访问控制（DAC）



大纲

- 基本概念
  - 操作系统的安全层次
  - 访问控制模型
  - SELinux
- 操作系统内部安全
  - 系统漏洞
  - 侧信道、隐秘信道
  - 缓存信道攻击
  - 缓存信道防御
  - Meltdown经典案例
- 恶意操作系统

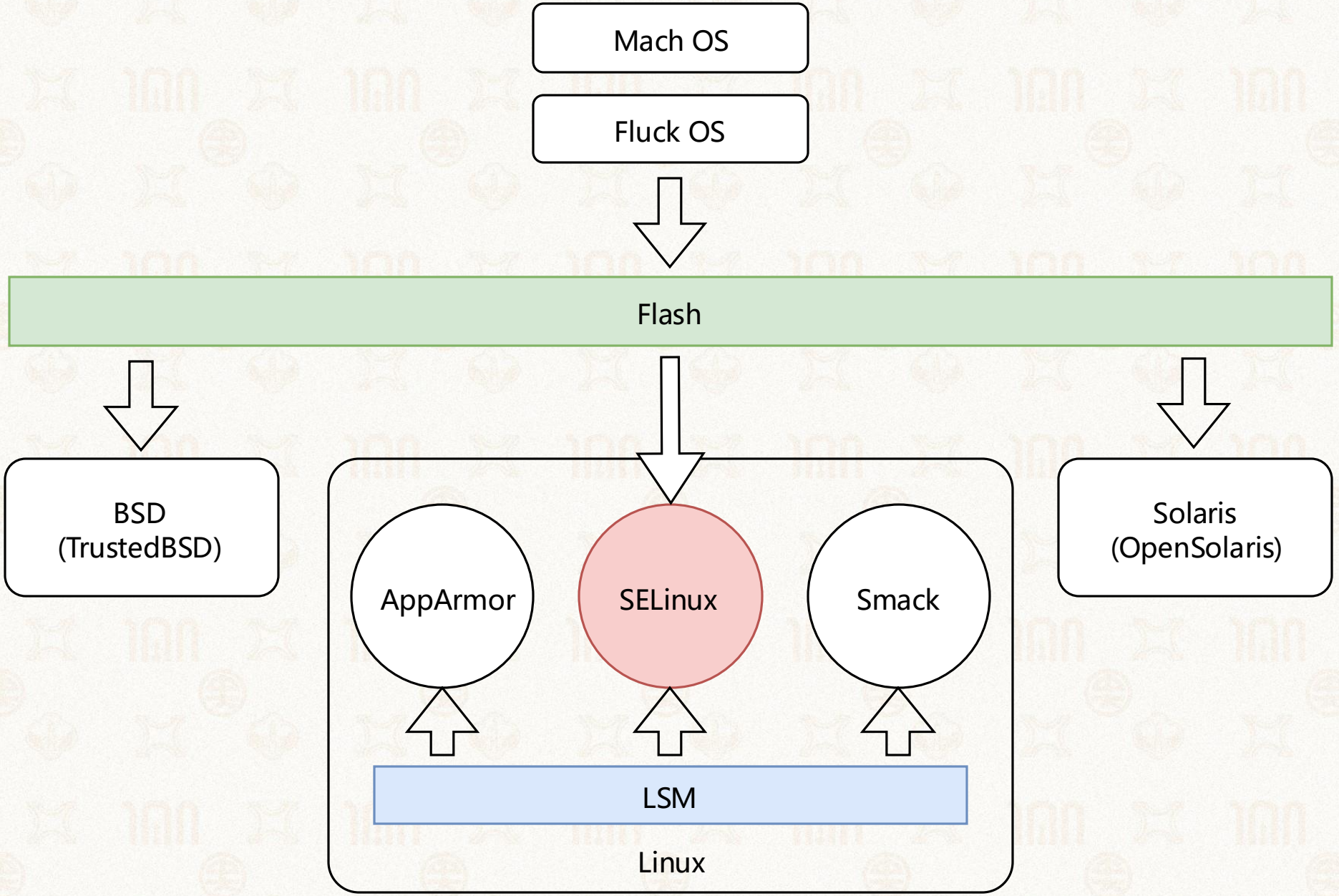
SELinux的历史

- SELinux，由NSA发起，2003年并入Linux
  - 是 Flask 安全架构在 Linux 上的实现
    - Flask 是一个 OS 的安全架构，可灵活提供不同的安全策略
  - 是一个 Linux 内核的安全模块（LSM）
    - 在Linux内核的关键代码区域插入了许多 hook进行安全检查

- SELinux 提供一套访问控制的框架
  - 支持不同的安全策略，包括强制类型访问（MAC）

https://selinuxproject.org/page/Main_Page

SELinux、Flask与LSM



SELinux引入的概念

- 用户（User）：指系统中的用户
  - 与 Linux 系统用户并没有关系

- 策略（Policy）：一组规则（Rule）的集合
  - 默认是"Targeted"策略，主要对服务进程进行访问控制
  - MLS （Multi-Level Security），实现了 Bell-LaPadula 模型
  - Minimum，考虑资源消耗，仅应用了一些基础的策略规则，一般用于手机等平台

- 安全上下文：是主体和对象的标签（Label）
  - 用于访问时的权限检查
  - 可通过"ls -Z"的命令来查看文件对应的安全上下文

SELinux的访问向量

- SELinux 将访问控制抽象为一个问题：
  - 一个 < 主体 > 是否可以在一个 < 对象 > 上做一个 < 操作 >

- AVC: Access Vector Cache
  - SELinux 会先查询AVC，若查不到，则再查询安全服务器
  - 安全服务器在策略数据库中查找相应的安全上下文进行判断

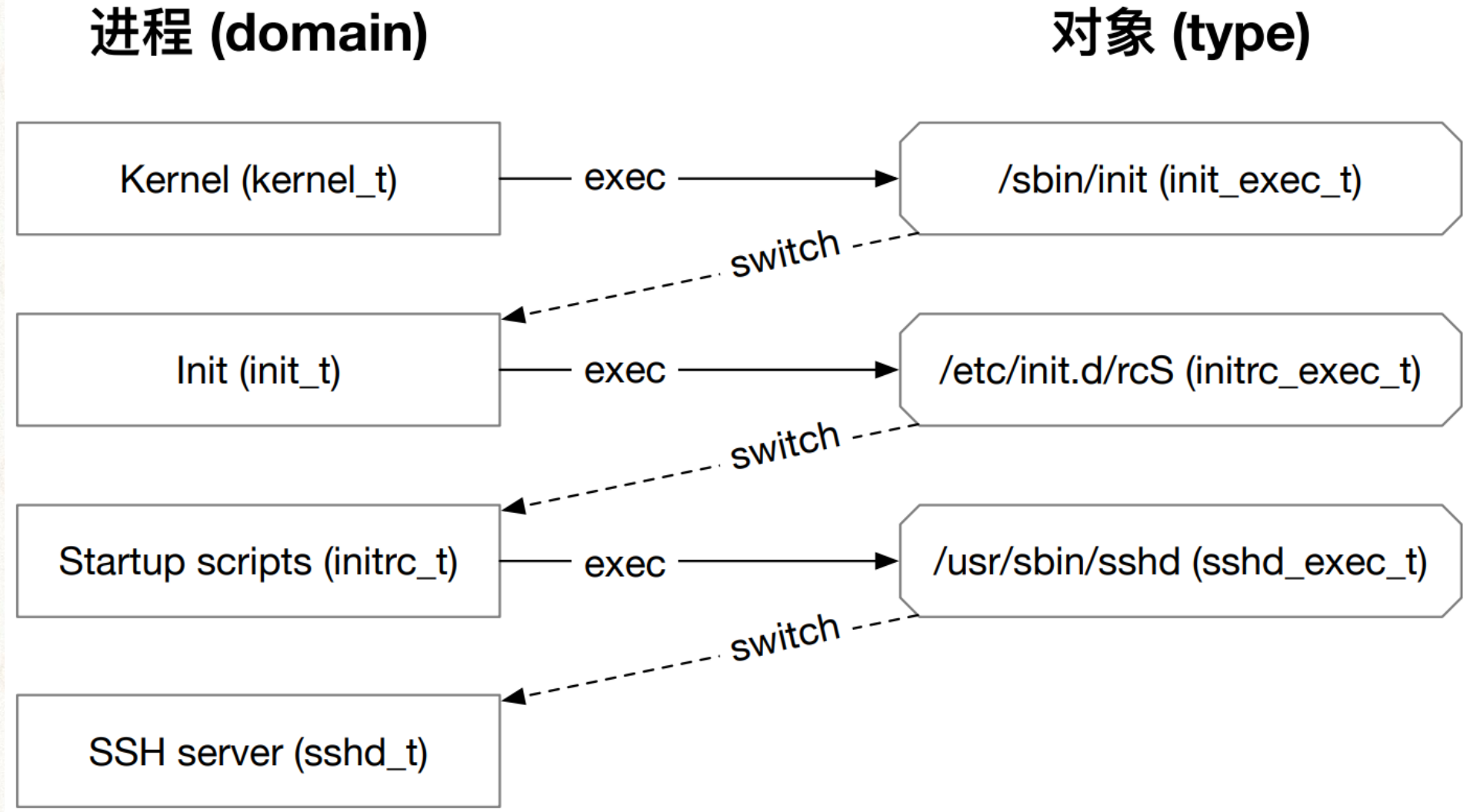
SELinux的安全上下文

- SELinux本质上是一个标签系统
  - 所有的主体和对象都对应了各自的标签

- 标签的格式 — 用户:角色:类型:MLS层级
  - 用户登录后，系统根据角色分配给用户一个安全上下文

- 类型（Type）用于实现访问控制
  - 每个对象都有一个 type
  - 每个进程的type称为 domain
    - 一个角色对应一个domain
    - 重要的服务进程被标记为特定的domain
    - 例如：/usr/sbin/sshd 的类型为 sshd_exec_t

进程的domain与对象的type



https://debian-handbook.info/browse/stable/sect.selinux.html

大纲

- 基本概念
  - 操作系统的安全层次
  - 访问控制模型
  - SELinux
- 操作系统内部安全
  - 系统漏洞
  - 侧信道、隐秘信道
  - 缓存信道攻击
  - 缓存信道防御
  - Meltdown经典案例
- 恶意操作系统

操作系统漏洞分类的三个角度

- 漏洞类型
  - 指攻击所利用的漏洞类型
  - 包括：栈/堆缓冲区溢出错误、整形溢出错误、空指针/指针计算错误、内存暴露错误、use-after-free 错误、 格式化字符串错误、竞争条件错误、参数检查错误、认证检查错误等

- 攻击模块
  - 指攻击所利用漏洞的所在的内核模块
  - 包括调度模块、内存管理模块、通信模块、文件系统、设备驱动等

- 攻击效果
  - 指攻击的目的或攻击导致的结果
  - 包括提升权限、执行任意代码、内存篡改、窃取数据、拒绝服务、破坏硬件等

整形溢出漏洞

- 在 32 位的系统中，一个无符号的 int 数在累加到 2^{32} 后，再加 1 就会变成 0
- 一个有符号的 int 数在累加到 2^{31} 后，再加 1 就会变成 -2^{31}

```
unsigned long count =  /* from user space */;
if(count > 1 << 30) {
    return -EINVAL;
}

table = vmalloc(sizeof(struct rps_dev_flow_table) + count * sizeof(struct rps_dev_flow));
//...
for(i = 0; i < count; i++) {
    table->flow[i] = /* ... */;
}
```

8字节

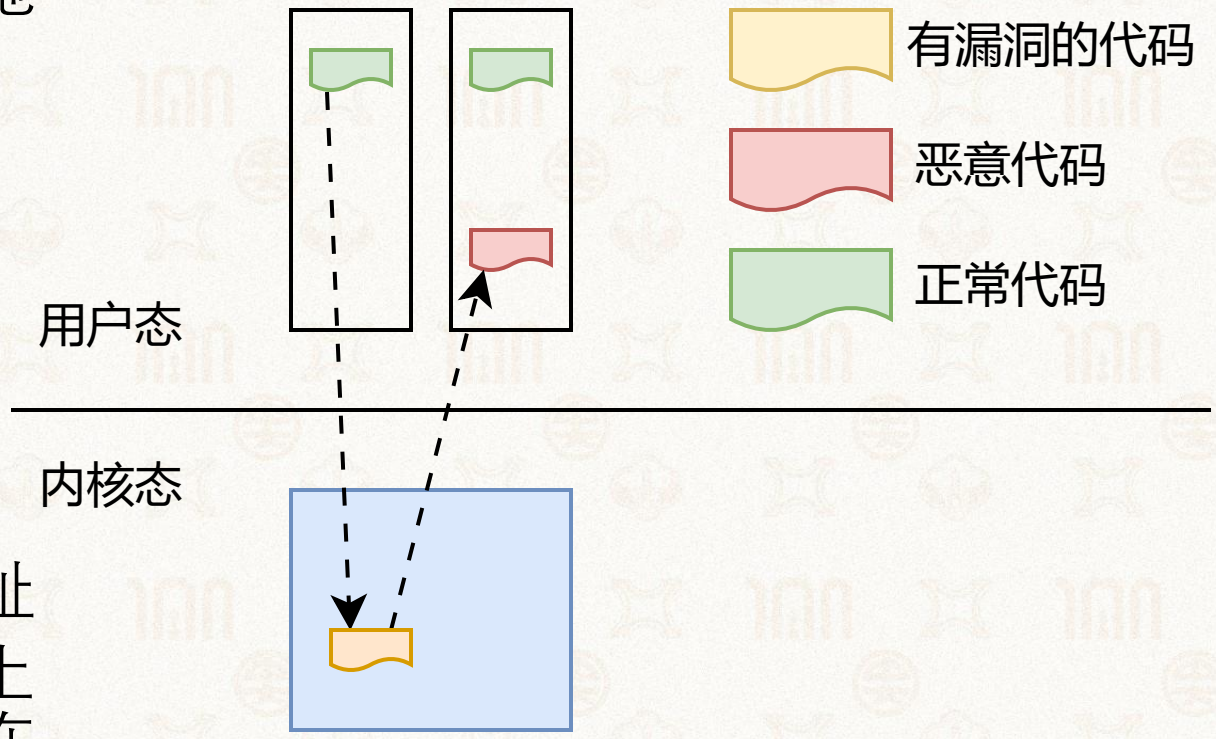
如果count = 2^30， 会发生什么?

会发生越界

Return-to-user攻击（ret2usr）

- 内核错误的运行了用户态的代码
  - 由于内核与应用程序共享同一个页表
  - 内核运行时可以任意访问用户态的虚拟地址空间
  - 内核可能执行位于用户态的代码

- 攻击者的常用方法
  - 先在用户态中初始加载一段恶意代码
  - 然后利用内核的某个漏洞， 修改内核中的某个函数指针指向这段恶意代码的地址
  - 也可以利用内核的栈溢出漏洞，覆盖栈上的返回地址为恶意代码的地址，使内核在执行 ret 指令时跳转到位于用户态的代码



ret2usr攻击的防御方法

- 方法一：仔细检查内核中的每个函数指针
  - 需对内核所有模块进行检查，很难做到 100% 的覆盖率
- 方法二：在陷入内核时修改页表，将用户态所有的内存都标记为不可执行
  - 由于修改页表后必须要刷新TLB才能生效，因此修改页表、刷新TLB，以及后续运行触发 TLB miss 都会导致性能下降
  - 在返回用户态之前必须将页表恢复，并再次刷掉 TLB，这样又会导致用户态执行时出现 TLB miss，因此对性能的影响非常大
- 方法三：硬件保证CPU处于内核态时不得运行任何用户态的代码
  - 如 Intel 的 SMEP（Supervisor Mode Execution Prevention）技术
  - ARM 同样有类似 SMEP 的技术，称为 PXN（Privileged eXecute-Never）

SMEP 不能完全解决 ret2usr：ret2dir

- 操作系统管理内存的方法"直接映射"
  - 将一部分或所有的物理内存映射到一段连续的内核态虚拟地址空间
  - 因此，同一块物理内存在系统中有多个虚拟地址
    - 例如，某个内存页分配给了应用程序，那么内核既可以通过应用程序的虚拟地址访问， 也可以通过直接映射的虚拟地址访问

- 基于直接映射的攻击，可绕过SMEP
  - 攻击者首先推算出位于用户态的恶意代码在内核直接映射区域的虚拟地址，然后在 ret2usr 攻 击中让内核跳转到该地址执行（内容依然为攻击者控制）
  - 攻击成功还有一个前提：直接映射区域必须是可执行的
    - 在 3.8.13 以及 之前的 Linux 版本，将直接映射区域的权限设置为了"可读-可写-可执行"
  - 这种利用直接映射区域的 ret2usr 攻击被称为"ret2dir"攻击

Rootkit：获取内核权限的恶意代码

- Rootkit 是指以得到 root 权限为目的的恶意软件
  - Rootkit 可以运行在用户态，也可以运行在内核态

- 用户态的Rootkit
  - 可以将自己注入到某个具有 root 权限的进程中，并接收攻击者的命令

- 内核态的Rootkit
  - 可以 hook 某个内核中的关键函数，从而在该函数被调用时触发运行
  - 可以是以内核线程的方式运行
  - 可以是修改内核中的系统调用表，用恶意代码来替换掉正常的系统调用

KASLR：内核地址布局随机化

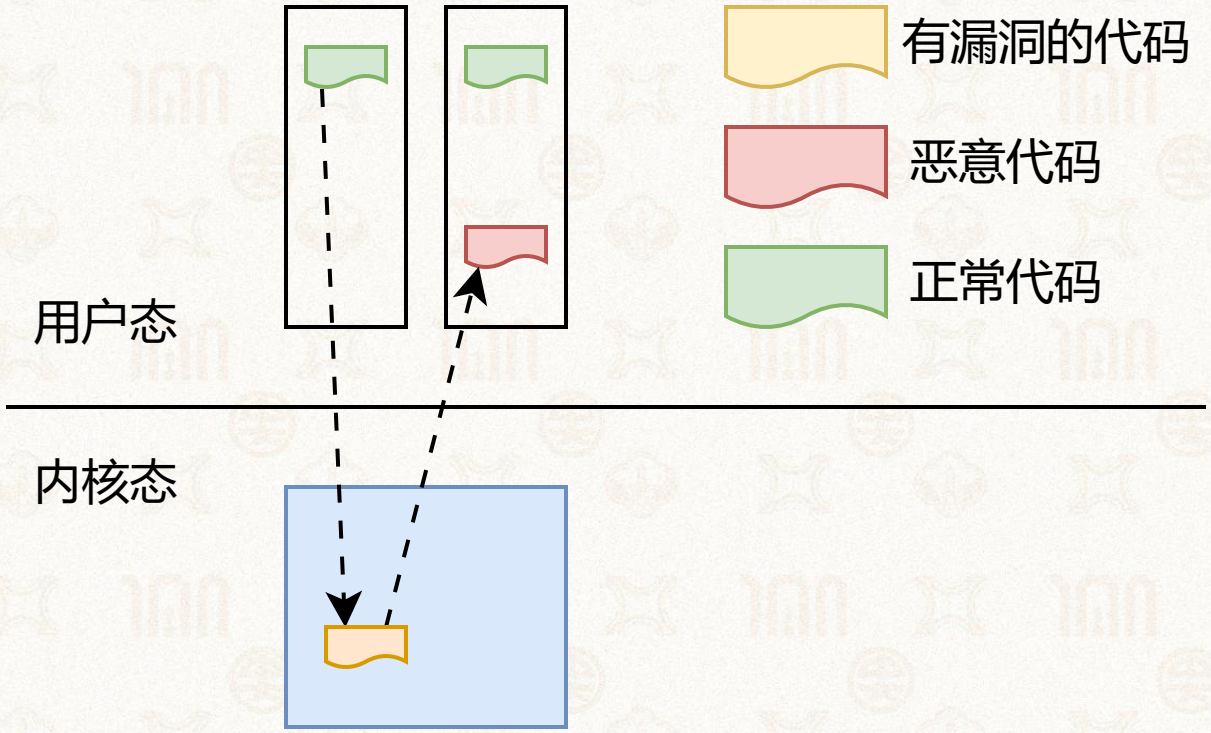
- Kernel address space layout randomization (KASLR)
- Address space layout randomization (ASLR)

ASLR 与 KASLR

- ASLR 通过随机化地址空间布局来提高系统攻击难度
- KASLR是对内核启用地址随机化

KASLR 可缓解 ret2dir 攻击

- 攻击者需要知道用户态恶意代码在内核中直接映射区域的地址
- KASLR 通过将内核的虚拟地址布局进行随机化，使攻击者准确定位内核地址的难度大大提升



大纲

- 基本概念
  - 操作系统的安全层次
  - 访问控制模型
  - SELinux
- 操作系统内部安全
  - 系统漏洞
  - 侧信道、隐秘信道
  - 缓存信道攻击
  - 缓存信道防御
  - Meltdown经典案例
- 恶意操作系统

什么是隐秘信道?

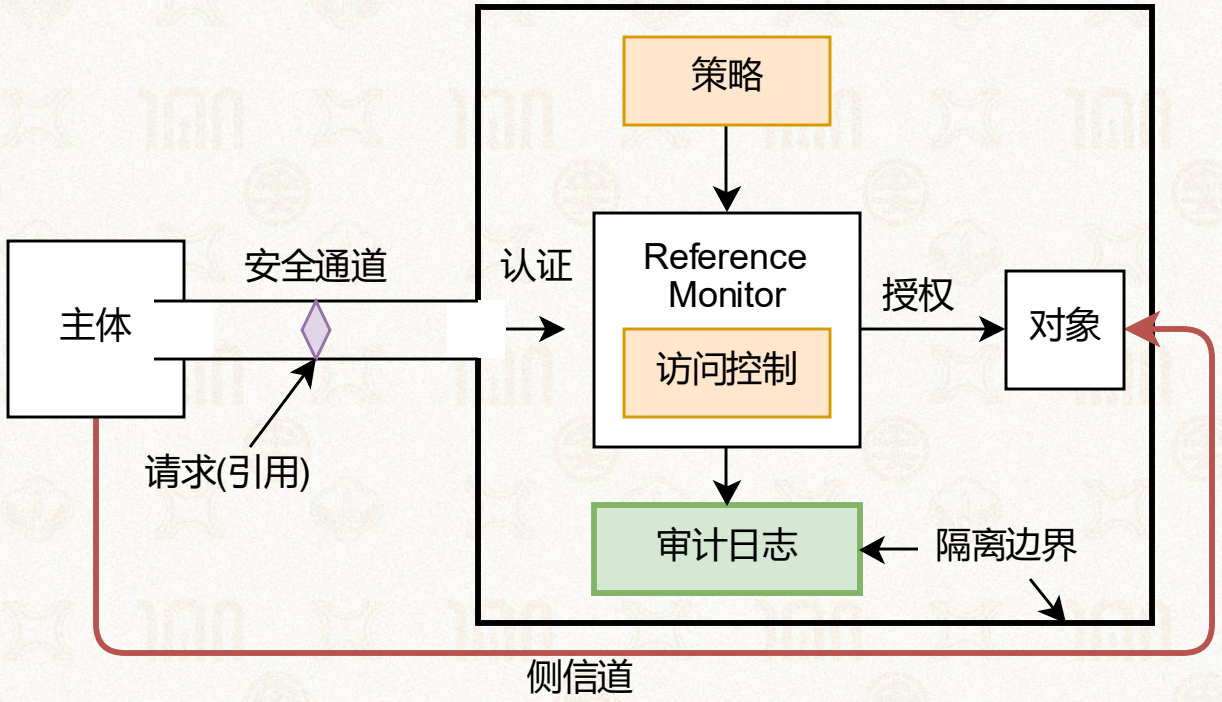
- 隐秘信道（Covert Channel）
  - 原本无法直接通信的两方，通过原本不被用于通信的机制进行数据传输
  - 常见的隐秘信道：时间、功耗、电磁泄露、声音等

- 例：消费记录的应用 A，在没有网络的情况下如何把数据发出去?
  - 假设有一个应用B运行在同一个手机
  - 若A可播放声音，B可录音，则A把数据编码为声音发送给B
  - 若A可打开闪光灯，B可摄像，则A把数据编码为光的闪烁长短与频率发送给B
  - 若A可震动，B可访问运动传感器，则A把数据编码为震动频率发送给B
  - 若B可访问CPU温度，则A可长时间运行计算密集代码，CPU升温表示1，反之为0
  - ...

侧信道与隐秘信道的关系

- 侧信道与隐秘信道很类似
  - 两者都使用类似的方式进行数据的传递

- 侧信道攻击和隐秘信道攻击的不同
  - 隐秘信道攻击：两方是互相串通的，其目的就是为了将信息从一 方传给另一方
  - 侧信道攻击：一方是攻击者，另一方是被攻击者，攻击者窃取被攻击者的数据
    - 即被攻击者无意通过侧信道泄露了自己的数据



时间信道(Timing Channel)

- 最常见的侧信道攻击方法
- 利用不同任务执行时间的差异推测细节

```
1 checkpw (user, passwd):
2   acct = accounts[user]
3   for i in range(0, len(acct.pw)):
4     if acct.pw[i] != passwd[i]:
5       return False
6   return True
```

如果第一位不相等，直接返回

时间信道(Timing Channel)

- 最常见的侧信道攻击方法
- 利用不同任务执行时间的差异推测细节

```
1 checkpw (user, passwd):
2   acct = accounts[user]
3   for i in range(0, len(acct.pw)):
4     if acct.pw[i] != passwd[i]:
5       return False
6   return True
```

如果第一位相等，for循环再执行第二句

时间信道(Timing Channel)

- 最常见的侧信道攻击方法
- 利用不同任务执行时间的差异推测细节

```
1 checkpw (user, passwd):
2   acct = accounts[user]
3   for i in range(0, len(acct.pw)):
4     if acct.pw[i] != passwd[i]:
5       return False
6   return True
```

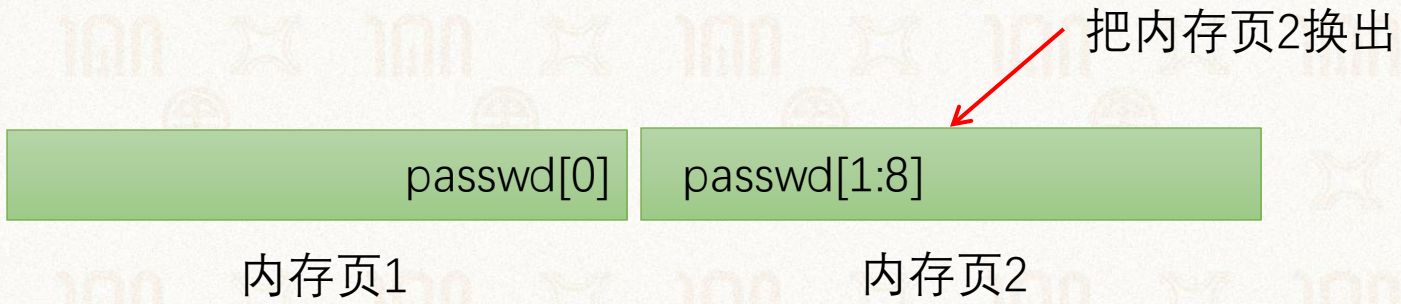
前面对的位数越多，for循环执行时间越长

时间信道(Timing Channel)

- 最常见的侧信道攻击方法
- 利用不同任务执行时间的差异推测细节

```
1 checkpw (user, passwd):
2   acct = accounts[user]
3   for i in range(0, len(acct.pw)):
4     if acct.pw[i] != passwd[i]:
5       return False
6   return True
```

- 将密码不同位存放在不同的内存页中



时间信道(Timing Channel)

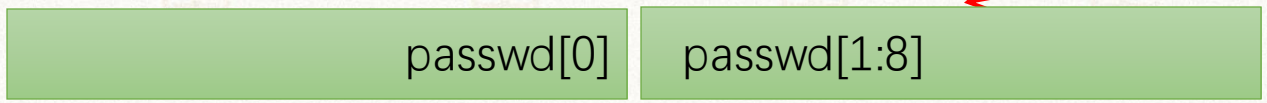
- 最常见的侧信道攻击方法
- 利用不同任务执行时间的差异推测细节

```
1 checkpw (user, passwd):
2   acct = accounts[user]
3   for i in range(0, len(acct.pw)):
4     if acct.pw[i] != passwd[i]:
5       return False
6   return True
```

- 如果passwd[0]猜错了，很快返回
- 如果passwd[0]猜对了，触发内存页换入操作 （变慢）

通过时间差异，可以判断第一位是否猜对

如果不需要读passwd[1]，就不需要换入



时间信道(Timing Channel)

- 最常见的侧信道攻击方法
- 利用不同任务执行时间的差异推测细节

```
1 checkpw (user, passwd):
2   acct = accounts[user]
3   for i in range(0, len(acct.pw)):
4     if acct.pw[i] != passwd[i]:
5       return False
6   return True
```

- 如果每一位有62种可能(大小写字母、数字),
  - 那么8位密码暴力破解需尝试 62^8 次

- 利用时间信道攻击，只需试验 $62 * 8$ 次就可以把密码偷到

大纲

- 基本概念
  - 操作系统的安全层次
  - 访问控制模型
  - SELinux
- 操作系统内部安全
  - 系统漏洞
  - 侧信道、隐秘信道
  - 缓存信道攻击
  - 缓存信道防御
  - Meltdown经典案例
- 恶意操作系统

缓存信道（Cache Channel）

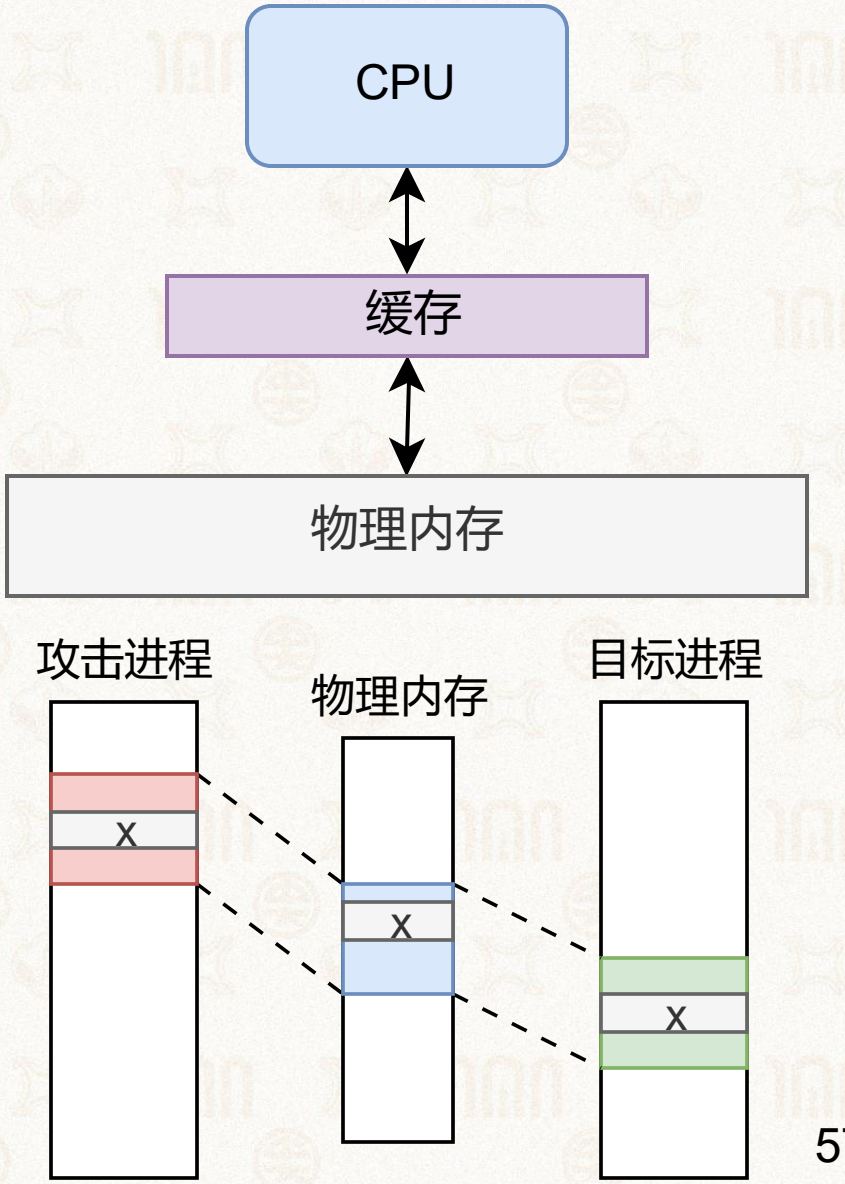
- 利用缓存的状态推测执行的信息
  - 例如：可根据 func_a 还是 func_b的代码在缓存中，判断 i 的值

```
if (i == 0)
    func_a();
else
    func_b();
```

- 常见的四种攻击方式
  - Flush+reload
  - Flush+flush
  - Prime+probe
  - Evict+time

缓存信道攻击：Flush + reload

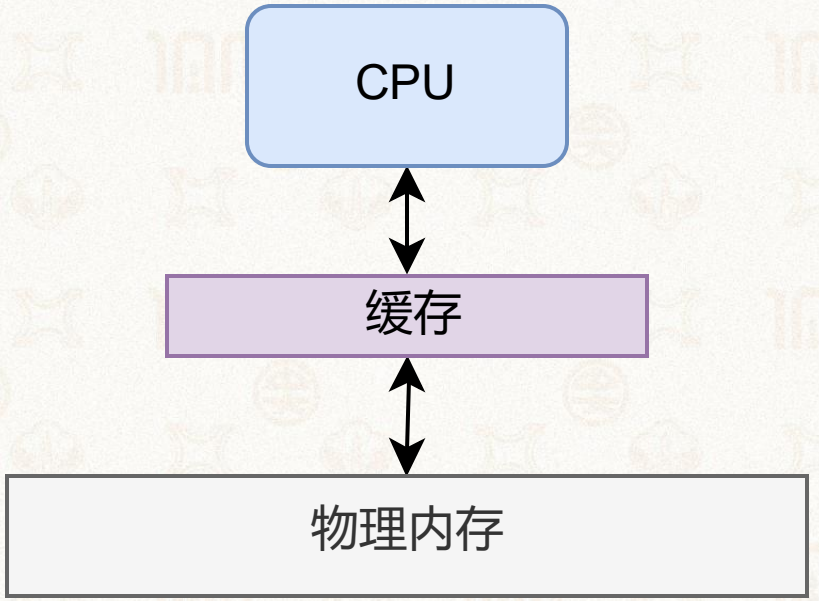
- 假设攻击进程和目标进程共享一块内存
  - 攻击者的目标是想知道目标进程是否访问了这块共享内存中的某个变量
- 第一步：攻击进程清空缓存

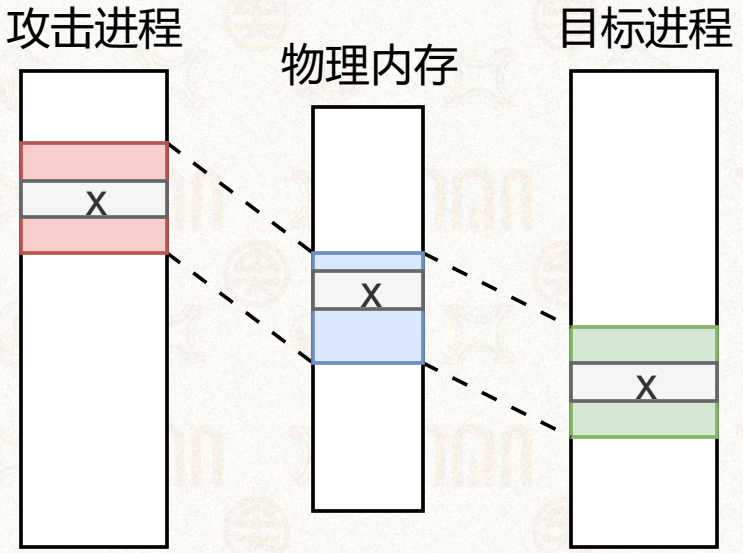


缓存信道攻击：Flush + reload

- 假设攻击进程和目标进程共享一块内存
  - 攻击者的目标是想知道目标进程是否访问了这块共享内存中的某个变量

- 第一步：攻击进程清空缓存
- 第二步：等待目标进程执行

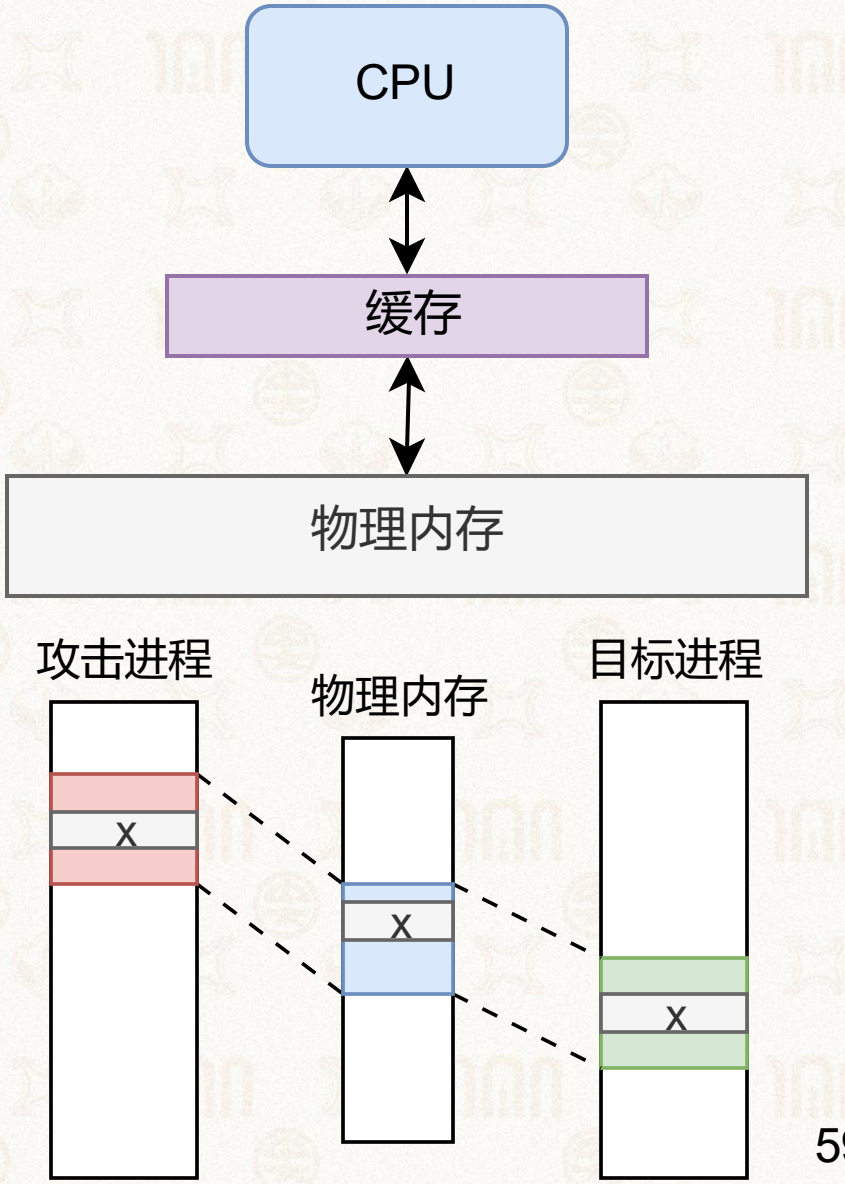




缓存信道攻击：Flush + reload

- 假设攻击进程和目标进程共享一块内存
  - 攻击者的目标是想知道目标进程是否访问了这块共享内存中的某个变量x

- 第一步：攻击进程清空缓存
- 第二步：等待目标进程执行
- 第三步：攻击进程访问变量x
  - 若访问时间长，则x不在缓存中
  - 说明目标进程没有使用x

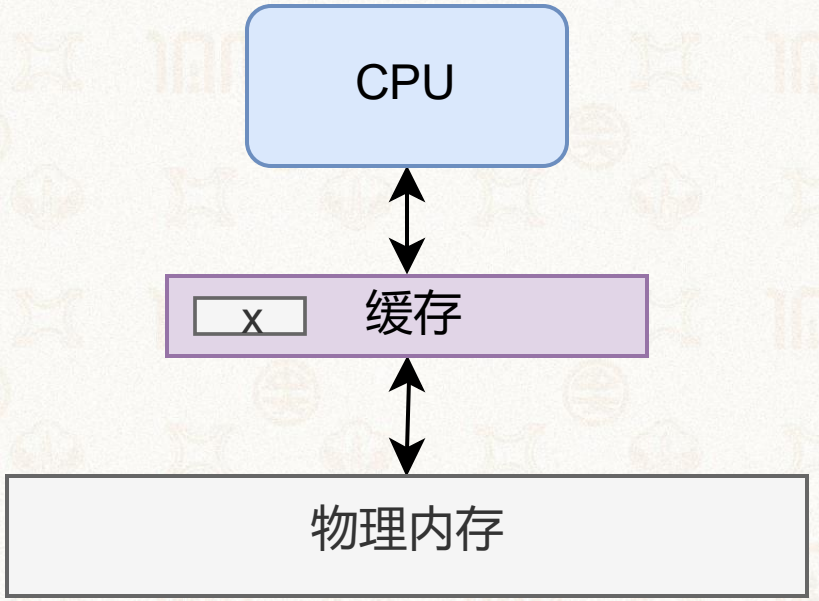


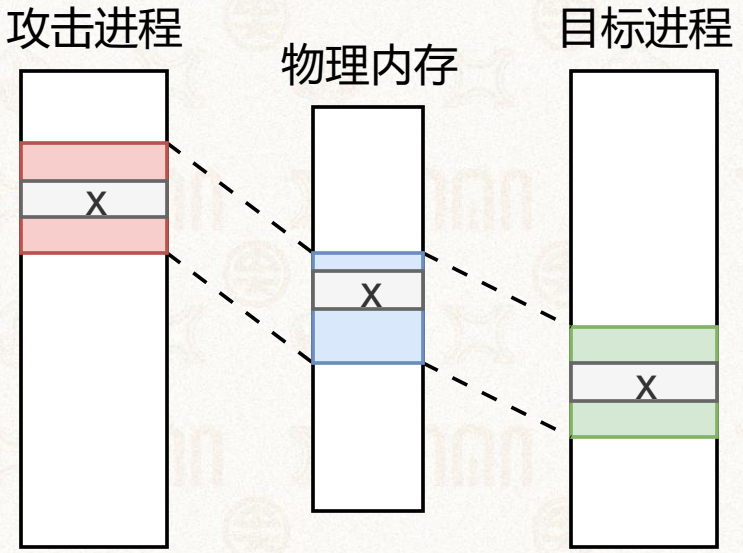
缓存信道攻击：Flush + reload

- 假设攻击进程和目标进程共享一块内存
  - 攻击者的目标是想知道目标进程是否访问了这块共享内存中的某个变量x

- 第一步：攻击进程清空缓存
- 第二步：等待目标进程执行
- 第三步：攻击进程访问变量x
  - 若访问时间短，则x在缓存中
  - 说明目标进程使用过x

- 通过观察变量读取速度(由缓存机制引发的差异)，推测某变量是否被使用过

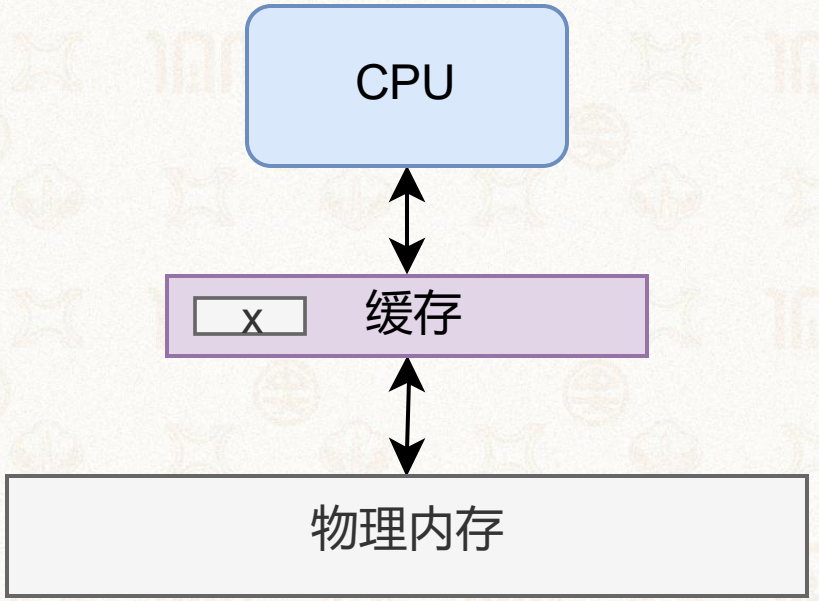


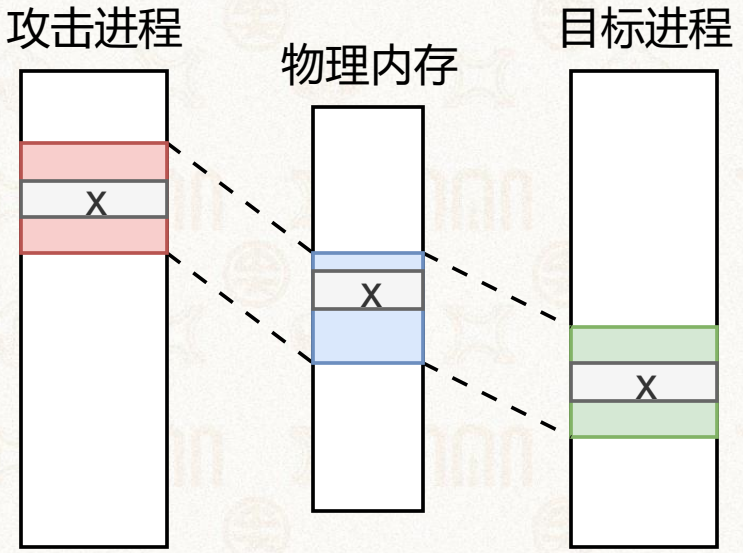


缓存信道攻击：Flush + flush

- 基于缓存刷新时间（如clflush）来推测数据在缓存中的状态

- 若数据在缓存中，那么clflush的执行时间相比数据不在缓存中要更长





防御侧信道/隐秘信道攻击

- 侧信道攻击很难被完全防御住，根本原因在于：共享
  - 当被攻击者在做了某个操作后，对系统整体产生了影响
  - 这个影响能够被使用同样系统的攻击者发现，那么就构成了一个最简单的侧信道：发现影响和没发现影响（即做了操作和没做操作）可以被编 码为 0 和 1

- 防御侧信道的根本方法：不共享
  - 将攻击者和被攻击者运行在完全隔离的物理主机，使其没有任何共享， 包括计算硬件、网络，甚至空间（光、温度、声音）
  - 更实际的方法是针对常见攻击进行防御，使计算不受外界影响

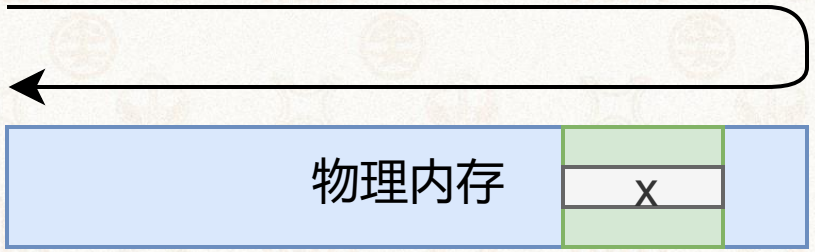
大纲

- 基本概念
  - 操作系统的安全层次
  - 访问控制模型
  - SELinux
- 操作系统内部安全
  - 系统漏洞
  - 侧信道、隐秘信道
  - 缓存信道攻击
  - 缓存信道防御
  - Meltdown经典案例
- 恶意操作系统

不经意随机访问内存（ORAM）

- ORAM 将访存行为与程序执行过程解耦
  - 攻击者即使能够观察到所有的访存请求，也无法反推出与程序执行相关的信息

- 最简单的实现：定时、定量、定位的访问方式
  - 无论实际是否有访存需求，均以固定周期，访问固定位置，每次访问固定的大小
  - 例如， CPU 顺序循环访问所有的有效内存区域
  - 程序按需获得真正想要访问的数据，
  - 若还没访问到则等待
  - 若已经访问过了则等待下次循环



- ORAM 会引入很大的额外负载
  - 产生大量的无效内存访 问，导致有效访存的吞吐率下降;另一方面
  - 访存需要等待一定的时刻，导致时延大幅度增加

大纲

- 基本概念
  - 操作系统的安全层次
  - 访问控制模型
  - SELinux
- 操作系统内部安全
  - 系统漏洞
  - 侧信道、隐秘信道
  - 缓存信道攻击
  - 缓存信道防御
  - Meltdown经典案例
- 恶意操作系统

Meltdown与Spectre攻击

- Meltdown（熔断）和 Spectre（幽灵）
  - 在 2017 年被发现，在 2018 年被公开

- 开创了一类新的侧信道攻击方式
  - 利用了 CPU 的预测执行机制
  - 几乎所有的主流 CPU 都受到了影响
    - 包括 Intel、 AMD和部分 ARM 处理器
  - 许多软件厂商不得不紧急打补丁并做出对应的防御措施

CPU预测执行机制

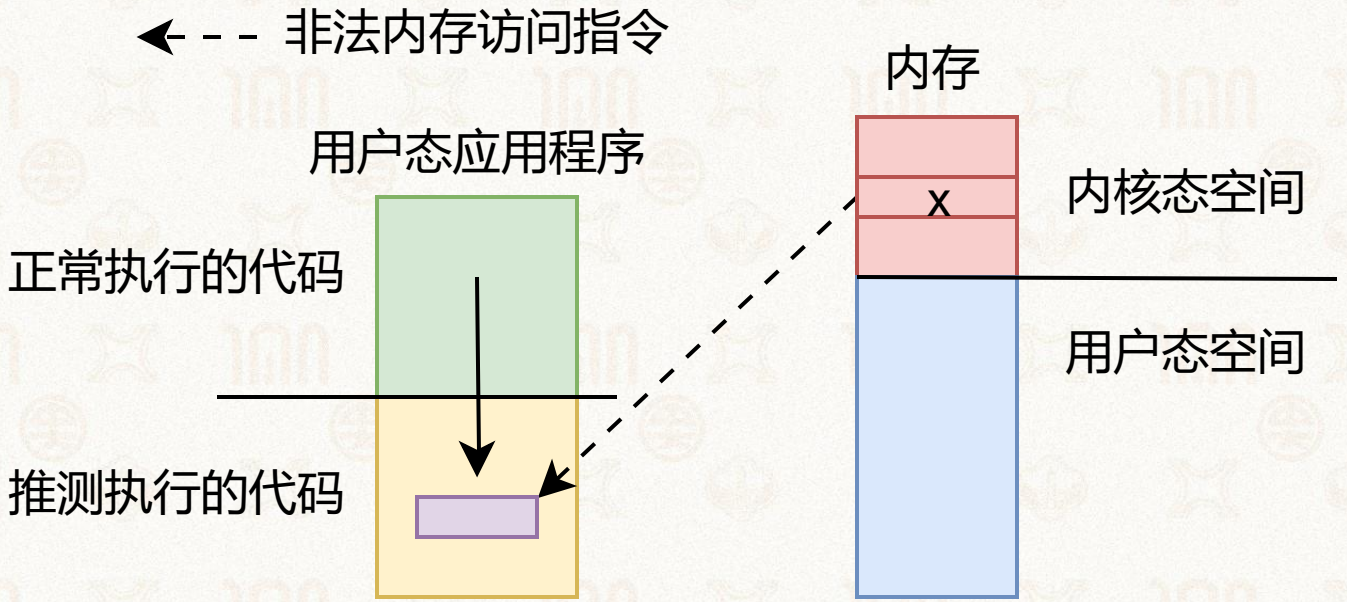
```
if (x != 0)
    y = a[10];
```

- CPU为了性能会进行预测执行
  - 当x的值为 0 时，对y的赋值并不会发生
  - 然而，若 CPU 在访问x时发生了阻塞（如 cache miss），CPU会先假设该条件成立，并实际去执行对y的赋值
    - 如果最后if的条件满足，则y的赋值已经完成，使性能得以提高
    - 若if条件不满足，CPU 会抛 弃对y的赋值，等价于没有执行过

- 然而，上述过程忽略了一个问题：
  - 数组 a[] 在预测执行的过程中被访问了一次——会有什么问题?

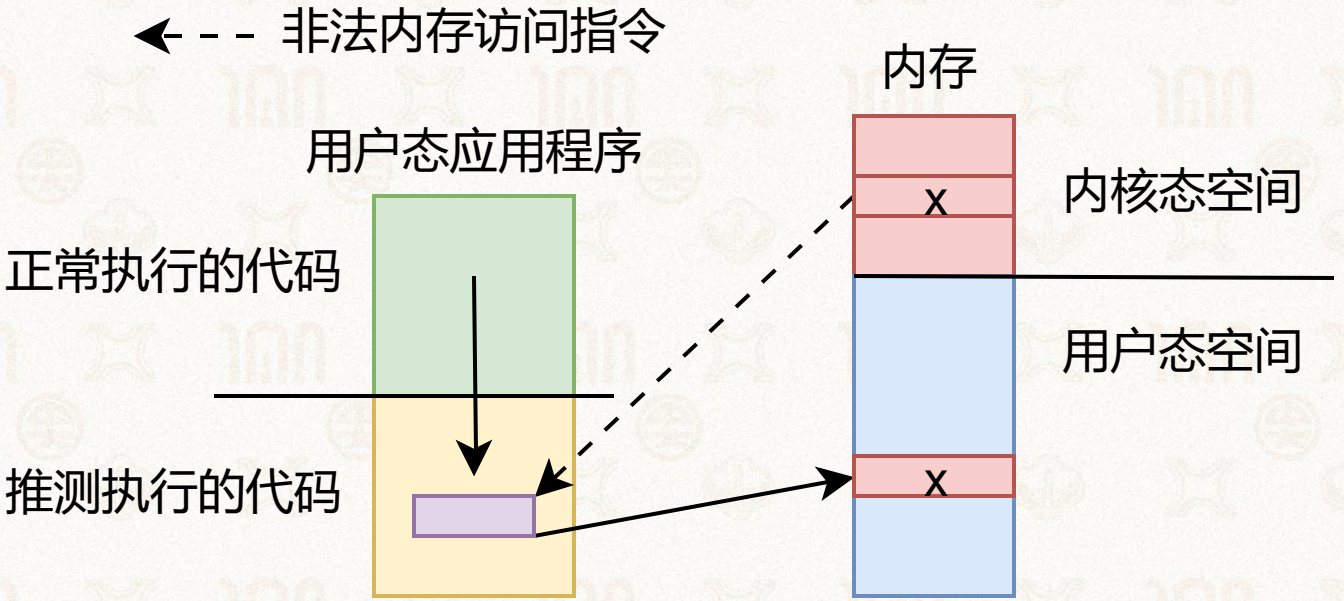
Meltdown攻击原理

- CPU的漏洞
  - 若让 CPU 预测执行一条跨权限非法内存访问的指令，CPU 不会进行权限检查；若最后预测条件不满足，CPU不会报错



Meltdown攻击原理

- CPU的漏洞
  - 若让 CPU 预测执行一条跨权限非法内存访问的指令，CPU 不会进行权限检查；若最后预测条件不满足，CPU不会报错
  - 攻击者可通过预测执行的方式，非法访问内核数据，然后根据内核数据改变缓存的状态，最后观测缓存状态反推出内核数据



Meltdown攻击原理

- CPU的漏洞
  - 若让 CPU 预测执行一条跨权限非法内存访问的指令，CPU 不会进行权限检查；若最后预测条件不满足，CPU不会报错
  - 攻击者可通过预测执行的方式，非法访问内核数据，然后根据内核数据改变cache状态，最后观测cache状态反推出内核数据

- 问：属于侧信道还是隐秘信道？
- 答：属于隐秘信道（Covert Channel）
- 攻击的两侧一边是构造cache状态，另一边是读取cache状态，两边都是攻击者控制；中间的墙，就是用户态与内核态的隔离

Meltdown攻击代码

```
; rcx = kernel address, rbx = probe array
xor rax, rax
retry:
mov al, byte [rcx]
shl rax, 0xc
jz retry
mov rbx, qword [rbx + rax]
```

来自 Meltdown 的论文

- 攻击能力很强
  - 把整个内核的内存都探测出来，最高可以达到 503KB/秒

KPTI：内核页表隔离

- 如何防御Meltdown漏洞？
  - 根本原因是硬件缺陷：预测执行时没有进行权限检查
  - 硬件修改不易，通过软件来弥补

- 在命名空间层，通过页表隔离
  - 使内核不再与应用共享虚拟地址空间，内核单独使用一个页表
  - 导致内核与应用状态切换时，需要新增页表的切换
    - 引入额外开销，包括TLB的刷新等
    - 部分应用性能降低可达30%

大纲

- 基本概念
  - 操作系统的安全层次
  - 访问控制模型
  - SELinux
- 操作系统内部安全
  - 系统漏洞
  - 侧信道、隐秘信道
  - 缓存信道攻击
  - 缓存信道防御
  - Meltdown经典案例
- 恶意操作系统

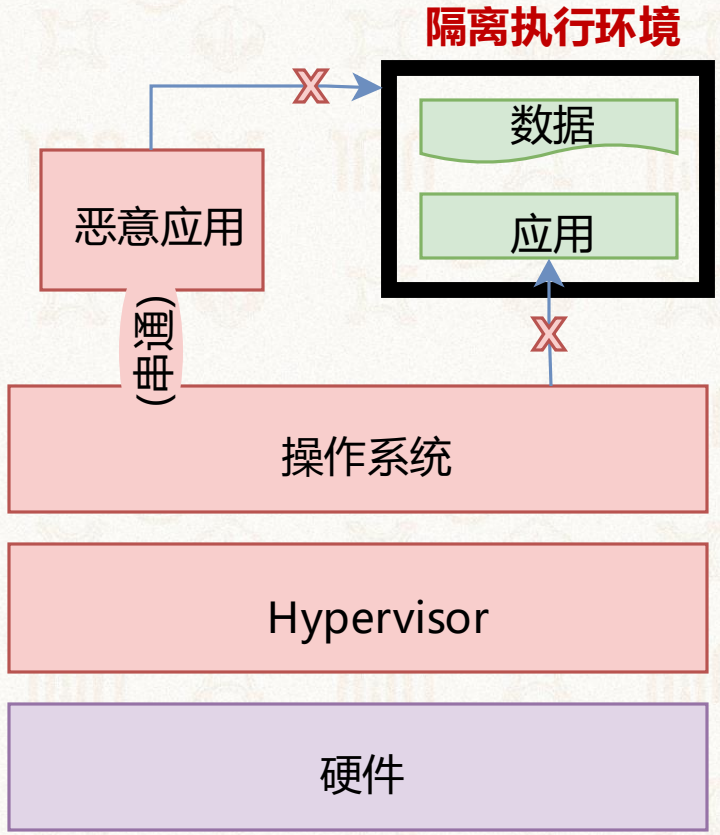
为什么要假设操作系统是恶意的?

- 系统的复杂性
  - 软件：恶意软件，OS本身可能存在漏洞
  - 硬件：外设越来越智能，本身可能存在漏洞，甚至是恶意构造
  - 环境：云计算环境、IoT设备，面临这更复杂多变的
  - 人：运维外包（如云计算等）导致接触计算机的人更复杂

- 一种更简单的威胁模型
  - “除了应用，别的都不可信”，即“自包含的信任”

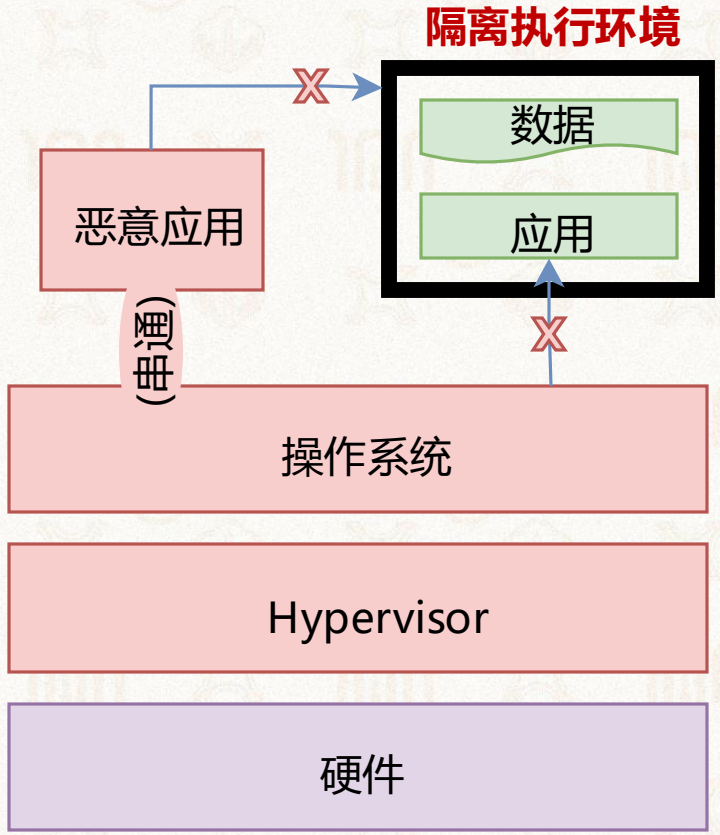
恶意操作系统如何攻击应用?

- 应用的攻击面
  - 同层：其他应用程序（如通过IPC）
  - 底层：操作系统、Hypervisor、硬件
- 操作系统窃取应用的数据
  - 操作系统控制着页表，可直接映射应用的内存并读取数据
- 操作系统改变应用的执行
  - 操作系统控制着页表，可直接在应用内部新映射一段恶意代码
  - 操作系统可任意改变程序的执行流



现实中的类似攻击

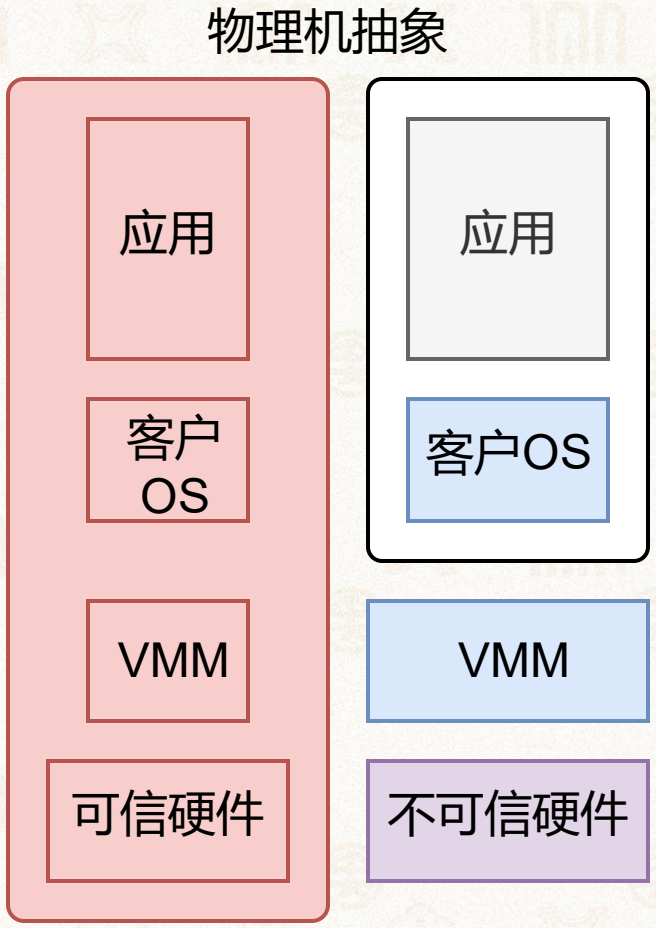
- 系统对应用的攻击
  - 攻击者获取具有root权限的bash（如利用sticky位程序的漏洞）
  - 攻击者获得insmod的权限，在内核中插入恶意的模块
  - 攻击者篡改内核的文件，下次启动后加载
  - 攻击者获得kexec的权限，动态执行另一个内核
  - 攻击者获得hypervisor的权限，从更底层发起攻击
  - 攻击者能够控制某个设备（如智能网卡），直接访问物理内存
  - ……



硬件隔离环境：Enclave

- 不信任CPU外的硬件
  - 包括内存（DRAM）、存储设备、网络设备、…
- 仅信任CPU
  - 包括cache、所有计算逻辑（无论如何，总得信任CPU吧...）
- Enclave（飞地）
  - 又称为可信执行环境，TEE（Trusted Execution Environment）

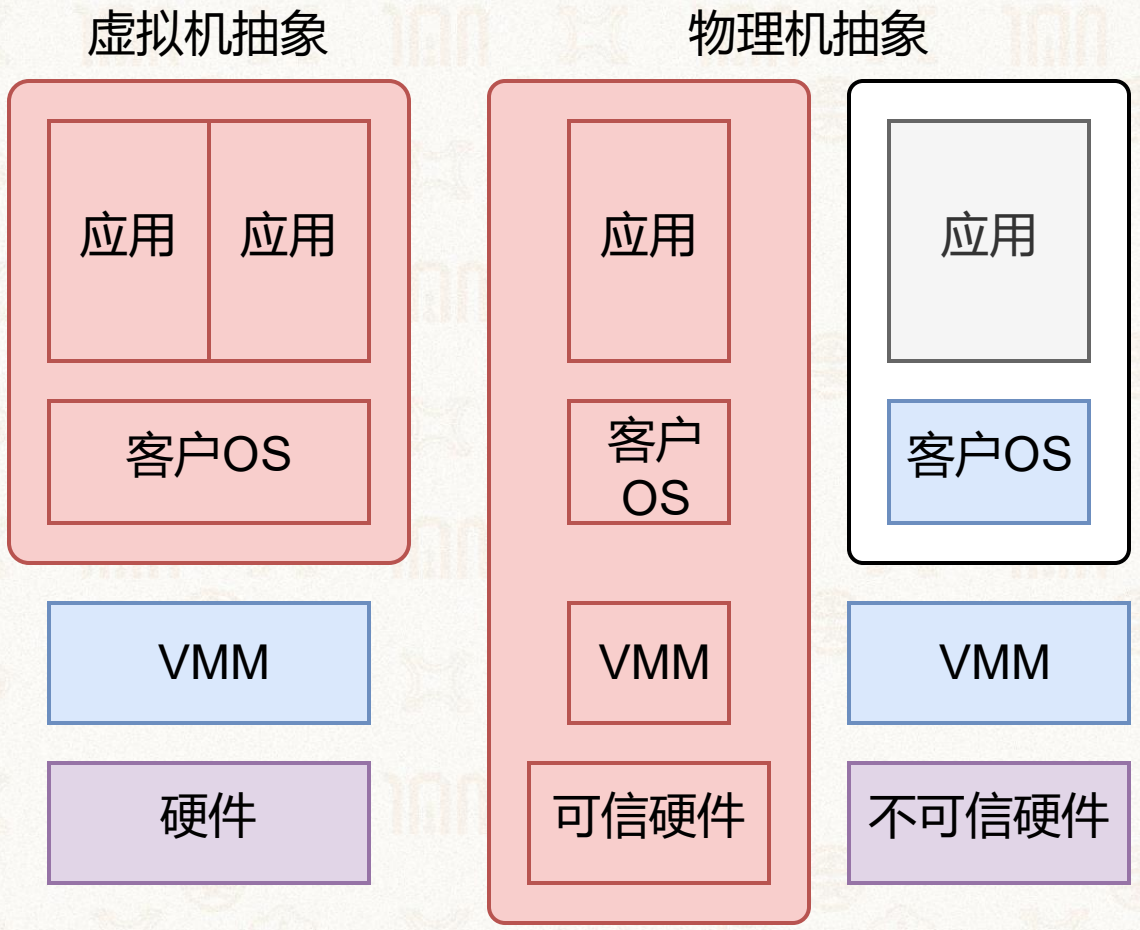
硬件提供不同粒度的隔离环境



2003年 ARM TrustZone

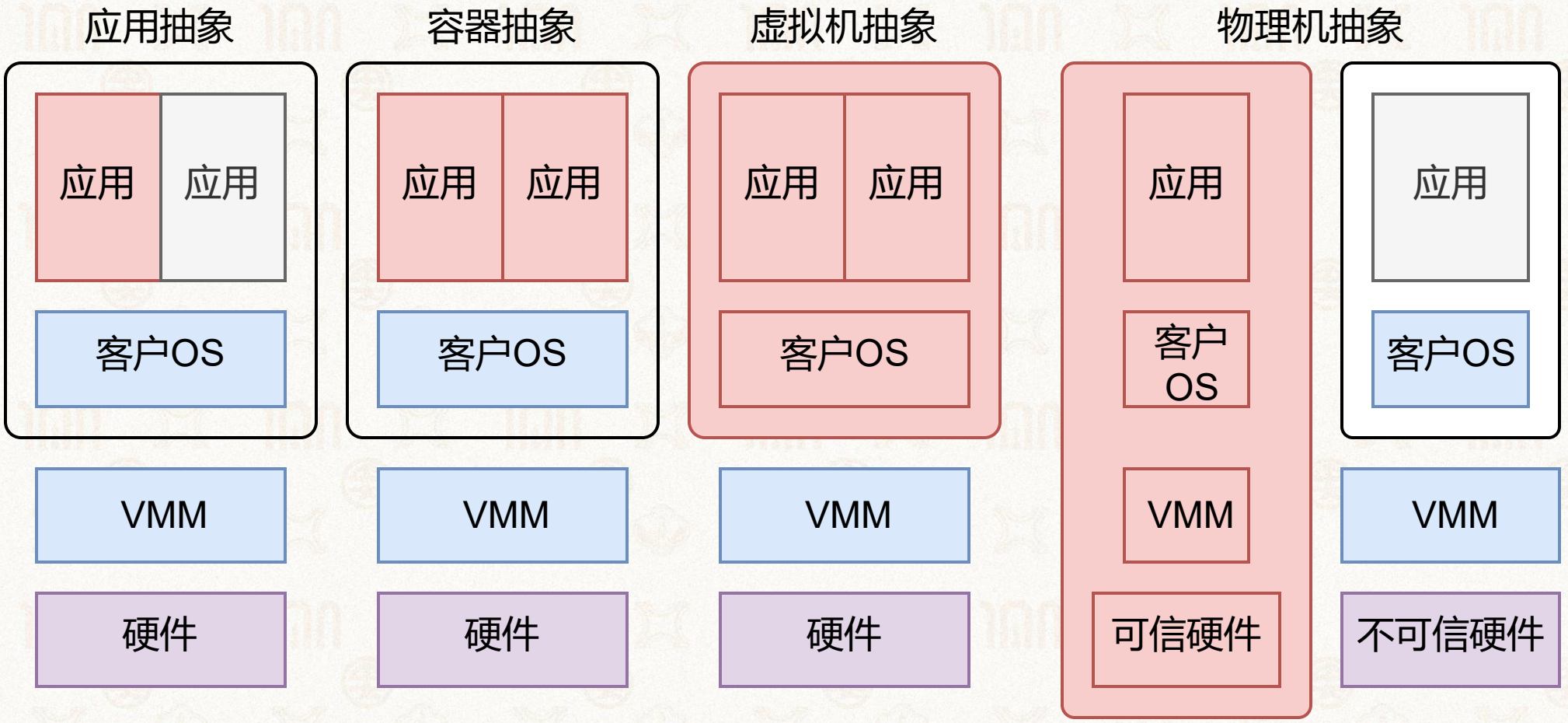
由于层次结构复杂，导致内部的软件漏洞不断

硬件提供不同粒度的隔离环境



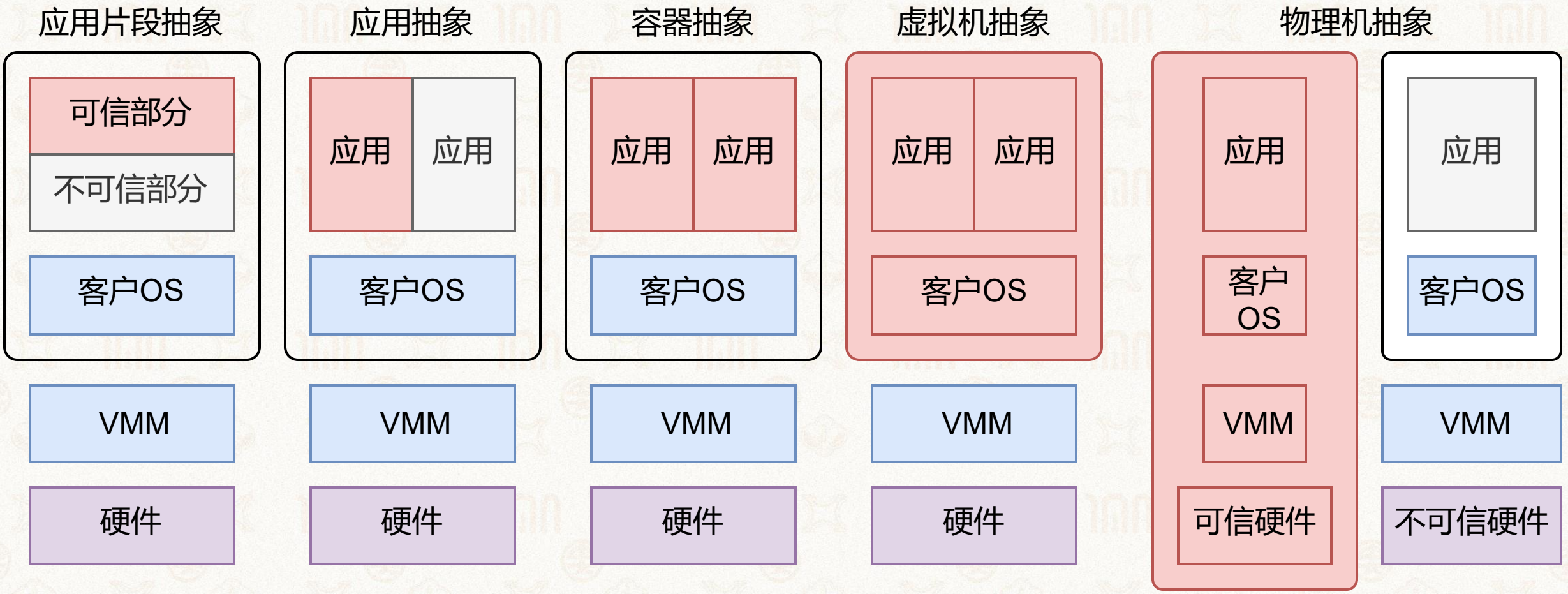
在内部复杂性与交互复杂性间寻找平衡点

硬件提供不同粒度的隔离环境



传统的操作系统隔离环境抽象，高度依赖OS语义，硬件不直接参与

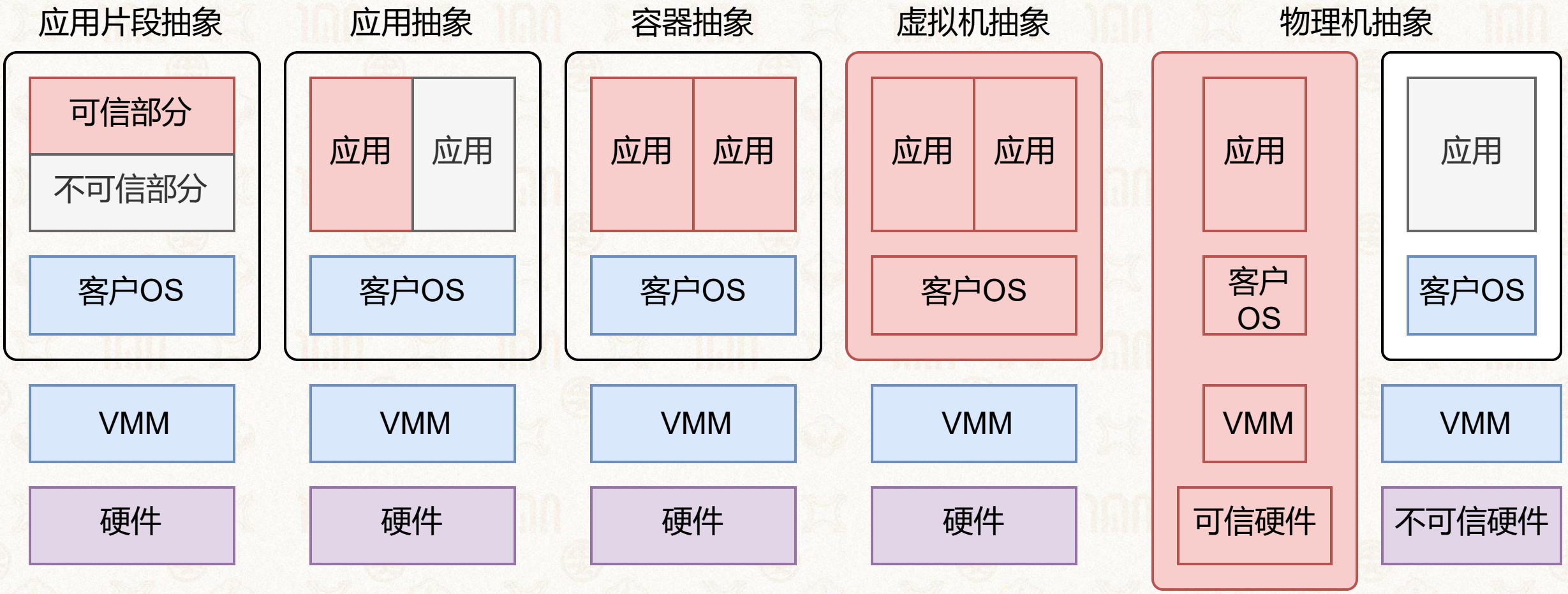
硬件提供不同粒度的隔离环境



2015年 Intel SGX

由硬件提供高层次接口，但交互漏洞频

硬件提供不同粒度的隔离环境

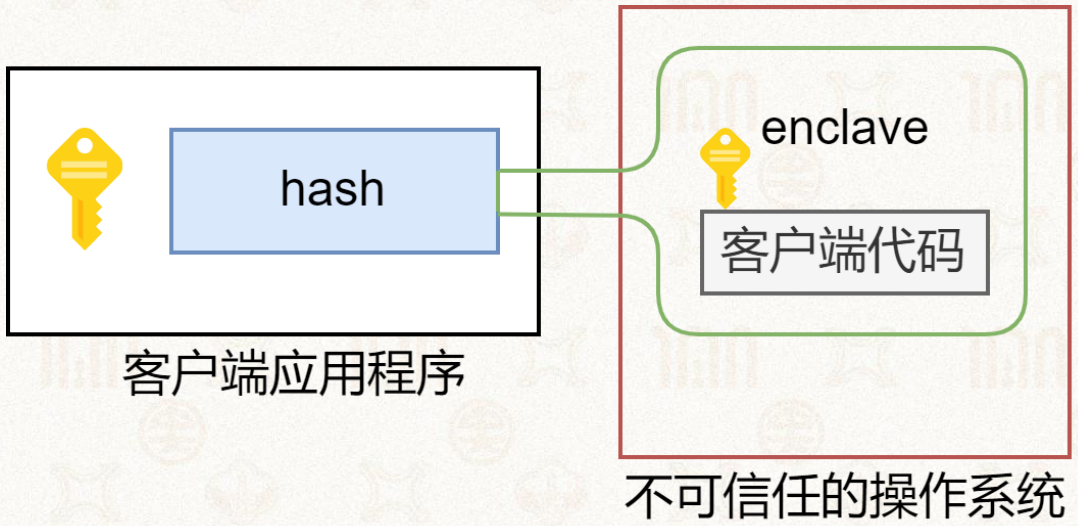


粒度细：内部简单，但交互复杂

粒度粗：交互简单，但内部复杂

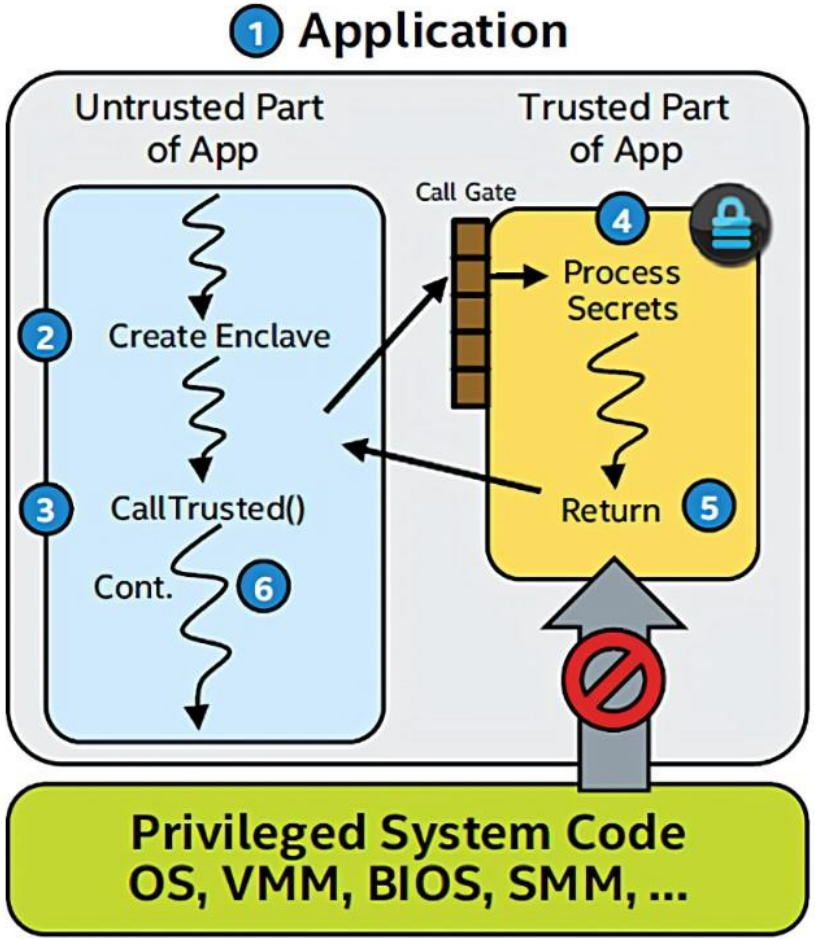
Enclave的隔离方法

- 基于权限控制
  - 使操作系统没有权限访问用户的数据
- 基于加密
  - 操作系统即使访问用户数据，也无法解密
- 基于权限控制+加密
  - 隔离防御软件攻击，加密防御硬件攻击



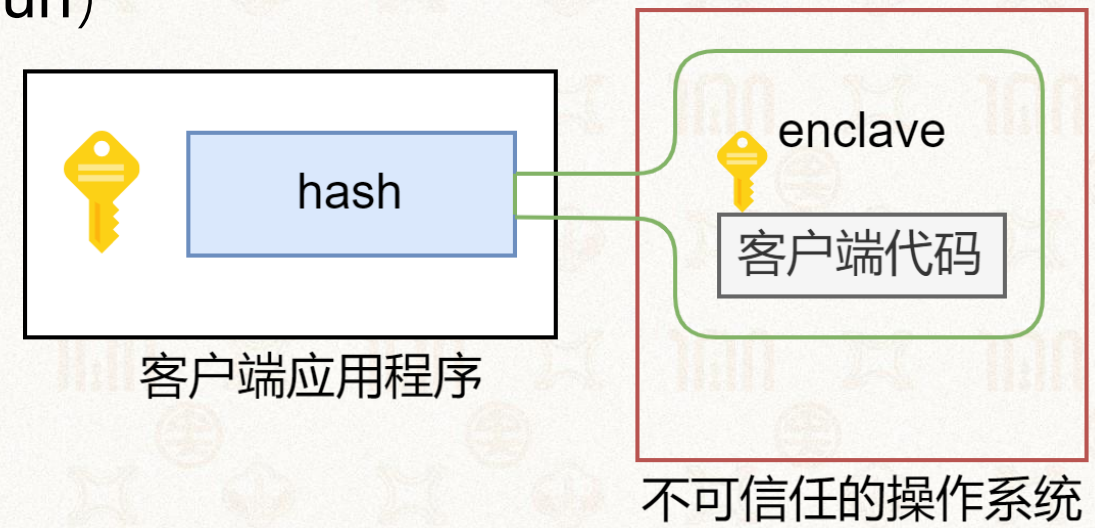
Enclave的隔离方法

- 基于预留的隔离（硬件）
  - 例如：PRM（Processor Reserved Memory）
  - CPU预留一部分物理内存，不提供给操作系统
- 基于页表的隔离（操作系统）
  - 例如：保证操作系统无法映射应用的物理内存页
  - 问题：页表是由操作系统自己管理的，监守自盗？
- 基于插桩的隔离（编译器）
  - 例如：SFI（Software Fault Isolation）
  - 在每次访存前插入边界检查，性能损失较大



Intel SGX

- SGX: Software Guard eXtension (硬件模块)
  - 2015年首次引入Intel Skylake架构
  - 保护程序和代码在运行时的安全（data in-run）
    - 其他安全包括：存储时安全和传输时安全



- 关键技术
  - Enclave内部与外部的隔离
  - 内存加密与完整性保护
  - 远程验证

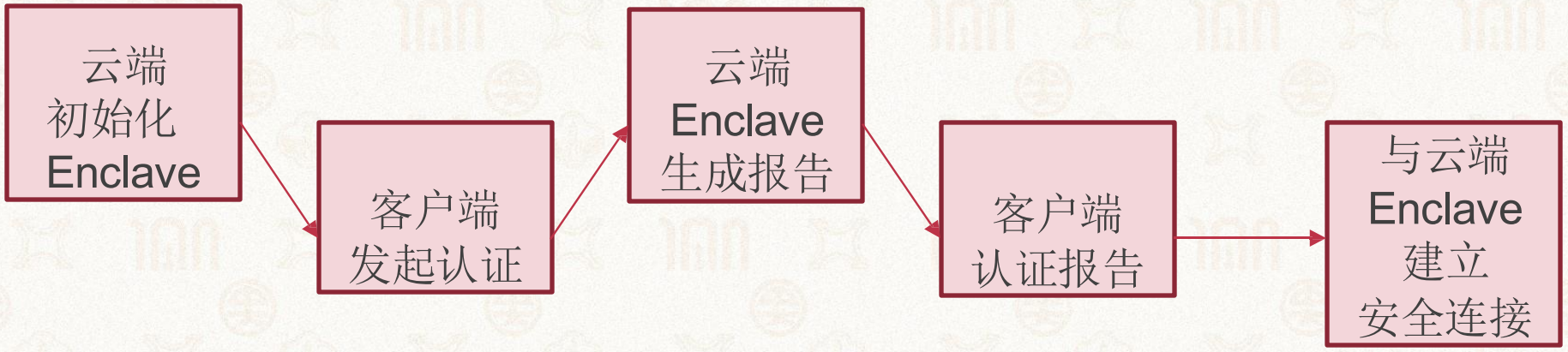
硬件内存加密与保护机制

- 硬件加密保护隐私性
  - CPU外皆为密文，包括内存、存储、网络等
  - CPU内部为明文，包括各级Cache与寄存器
  - 数据进出CPU时，由进行加密和解密操作

- 硬件Merkle Tree(默克尔树)保护完整性
  - 对内存中数据计算一级hash，对一级hash计算二级hash，形成树
  - CPU内部仅保存root hash，其它hash保存在不可信的内存中
  - 当内存中的数据被修改时，更新Merkle Tree

远程验证（Remote Attestation）

- 要解决的问题：如何远程判断某个主体是Enclave？
  - 例如，如何判断某个在云端的服务运行环境是安全的
  - 必须在认证之后，再进行下一步的操作，例如发送数据

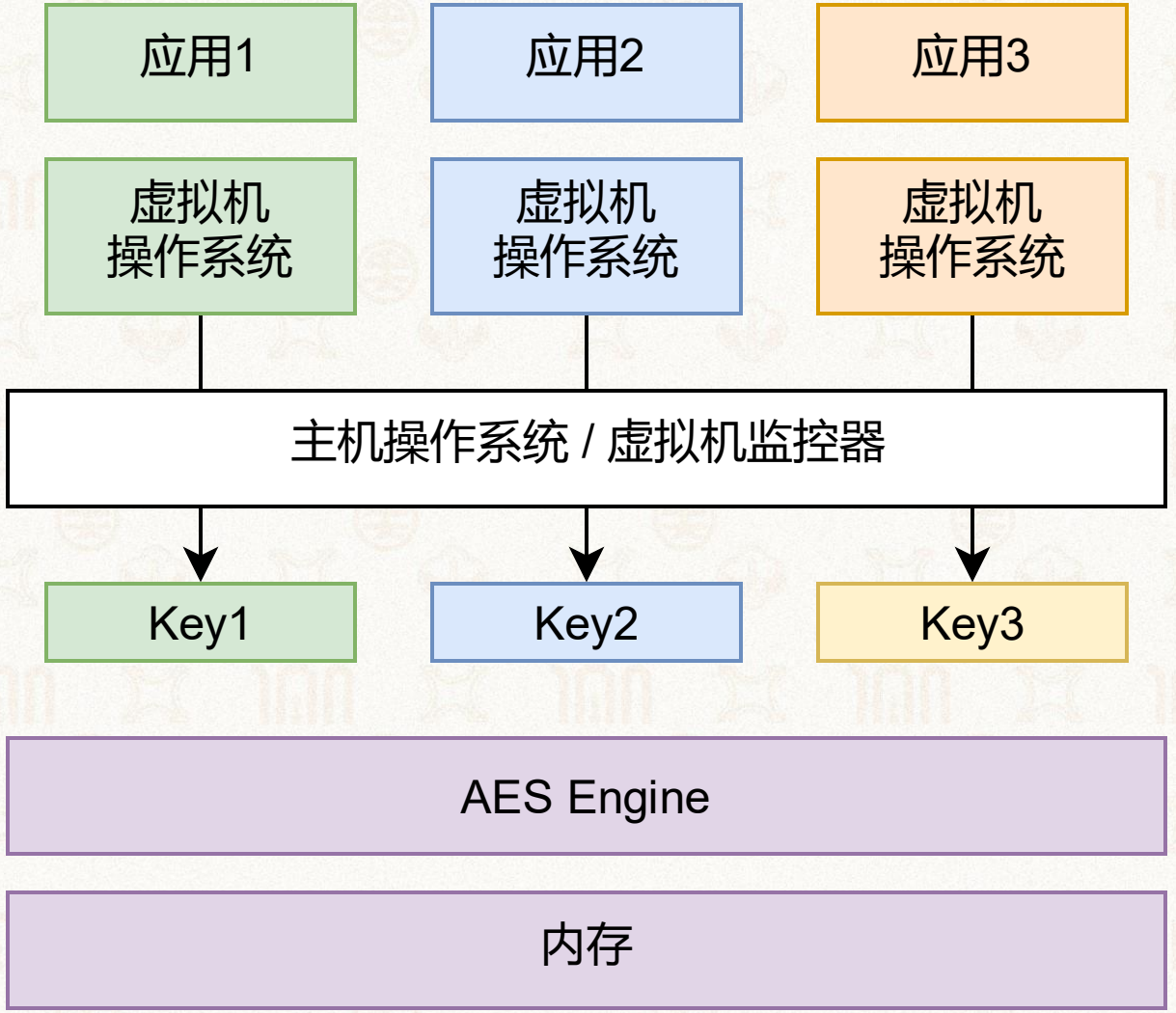


AMD SEV (安全加密虚拟化)

- AMD Secure Encrypted Virtualization (SEV)

- 以虚拟机为粒度的Enclave
  - 对不同的虚拟机进行加密
  - 每个虚拟机的密钥均不相同
  - Hypervisor有自己的密钥

- 安全模型的缺陷
  - 依然部分依赖Hypervisor
  - 如：为VM设置正确的密钥



Enclave隔离的不足

- 仅靠隔离是不够的，还需要考虑交互安全
  - Enclave依然需要OS提供服务：调度、系统调用、资源分配...
  - 即使隔离，OS依然可能发起的攻击包括
    - 接口攻击：合法的系统调用返回错误的值
      - 例：malloc返回指向栈的地址，导致内部自己破坏掉栈
    - DoS攻击：拒绝分配计算资源（恶意调度）

- 依然受到侧信道等攻击的威胁
  - Spectre、L1TF

从Intel新酷睿处理器放弃SGX看机密计算技术的发展: https://zhuanlan.zhihu.com/p/460081164

大纲

- 基本概念
  - 操作系统的安全层次
  - 访问控制模型
  - SELinux
- 操作系统内部安全
  - 系统漏洞
  - 侧信道、隐秘信道
  - 缓存信道攻击
  - 缓存信道防御
  - Meltdown经典案例
- 恶意操作系统

谢谢

微信：suyuxin
钉钉：苏玉鑫
B站：https://space.bilibili.com/502854403
软工集市课程专区：https://ssemarket.cn/new/course
匿名提问箱：https://suask.me/ask-teacher/106/苏玉鑫