

第4讲

安全体系结构 (2)

大纲

- The Confinement Principle约束原理
- 系统调用介入(System Call Interposition)
- 基于虚拟机的隔离
- 软件故障隔离(Software Fault Isolation)

运行不可信代码

经常需要运行不可信代码:

- **来自不可信网站的程序:**

- 应用, 扩展, 插件, 媒体播放器的编码解码器

- **暴露的应用程序:** pdf viewers, outlook

- **遗留守护进程:** sendmail, bind

- **蜜罐(honeypots)**

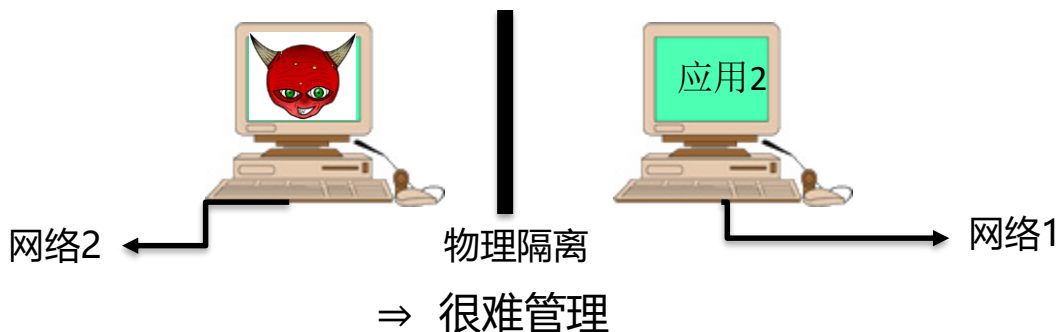
目的: 如果应用程序有“恶意行为” ⇒ kill 掉

方法: Confinement

Confinement: 确保恶意行为的应用程序不会伤害系统的其余部分

可以在不同层次实现:

- **硬件**: 在隔离的硬件上运行应用程序

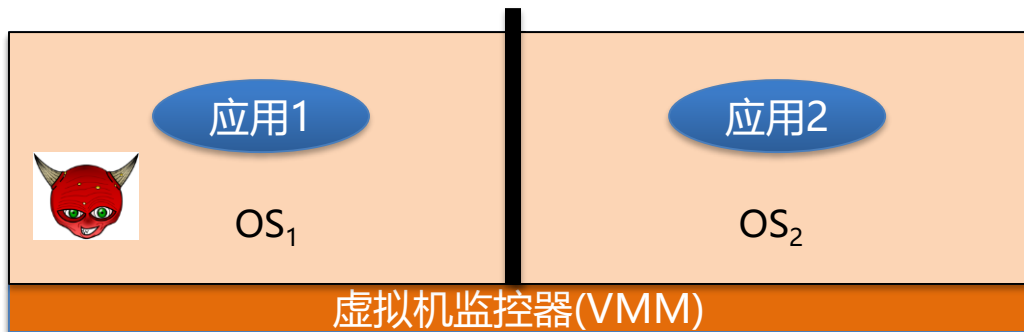


方法: Confinement

Confinement：确保恶意行为的应用程序不会伤害系统的其余部分

可以在不同层次实现：

- **虚拟机**：在单机上隔离OS



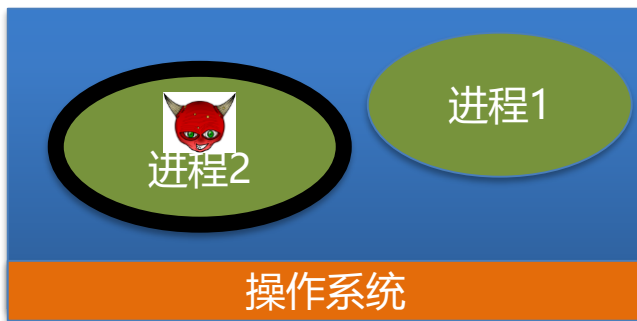
方法: Confinement

Confinement : 确保恶意行为的应用程序不会伤害系统的其余部分

可以在不同层次实现:

- **进程**: 系统调用介入 (System Call Interposition)

在操作系统中隔离进程



方法: Confinement

Confinement: 确保恶意行为的应用程序不会伤害系统的其余部分

可以在不同层次实现:

- **线程**: 软件故障隔离(SFI, Software Fault Isolation)
 - 隔离线程共享的地址空间
- **应用程序**: 例如, 浏览器的沙箱机制

Confinement的实现

关键组件: 引用监控器(reference monitor)

- 仲裁来自应用程序的请求
 - 实现保护策略
 - 执行隔离和限制
- 必须 始终 被调用:
 - 每个应用请求必须被仲裁处理
- 防篡改(Tamperproof):
 - 引用监控器不能被kill掉
 - 如果引用监控器被kill掉, 被监控的进程也会被kill掉
- 小, 小到足以进行分析和验证

chroot

chroot提供Change Root的功能，改变程序执行时所参考的根目录位置

作用：

1. 限制chroot的使用者所能执行的程序，如setuid的程序，或是会造成 Load 的Compiler等等
2. 防止使用者存取某些特定档案，如/etc/passwd
3. 防止入侵者 /bin/rm -rf /
4. 提供guest服务以及处罚恶意的使用者
5. 增进系统的安全

chroot

- chroot带来的好处：
 - 增强了系统的安全性，**限制用户**（即使是root）**权限**；
 - 建立一个与原系统**隔离的系统目录结构**，方便用户的开发；
 - 切换系统的根目录位置，引导 Linux 系统启动以及急救系统等。

chroot

使用方法：(必须使用 root 权限)

```
chroot /tmp/guest  
su guest
```

root 目录 “/” 现在是 “/tmp/guest”
EUID 设置为 “guest”

现在 “/tmp/guest” 被添加到 Jail 中应用程序可以访问到的文件系统

open(“/etc/passwd”, “r”) ⇒
open(“/tmp/guest/etc/passwd”, “r”)

⇒ 应用程序不能访问 Jail 外的文件

Jailkit

- 官方介绍:
 - Jailkit is a set of **utilities** to limit user accounts to specific files using `chroot()` and or specific commands.
 - Setting up a **chroot** shell, a shell limited to some specific command, or a daemon inside a `chroot` jail is a lot easier and can be automated using these utilities.

Where to find: <https://olivier.sessink.nl/jailkit/>

How to use it: <http://www.binarytides.com/setup-jailed-shell-jailkit-ubuntu/>

Jailkit

- jailkit 项目：自动构建文件、库以及jail环境中需要的目录
 - jk_init： 创建 jail 环境
 - jk_check： 检查 jail 环境的安全问题
 - 检查任何修改的程序
 - 检查全部可写的目录
 - jk_cp： copy文件到jail中
 - jk_lsh： 在jail中使用受限的shell

从jails中逃逸

逃逸方法: 相对路径

open(“../../etc/passwd”, “r”) ⇒

open(“/tmp/guest/../../etc/passwd”, “r”)

mkdir(d); chroot(d); cd ../../../; chroot(.)

Many ways to escape jail as root

- Create device that lets you access raw disk
 - chroot jail不会限制网络访问
- Send signals to non chrooted process
- Reboot system
- Bind to privileged ports

Freebsd jail

比简单的chroot更强大的机制

运行: **jail jail-path hostname IP-addr cmd**

- 调用加固的 chroot (避免 “../..” 逃逸)
- 只能绑定到具有指定IP地址和授权端口的套接字
- 只能与jail中的进程通信
- root是受限的, 比如: 不能加载内核模块

https://en.wikipedia.org/wiki/FreeBSD_jail

http://blog.sina.com.cn/s/blog_5d239b7f01019q58.html

不是所有的程序都可以在jail中运行

可以在jail中运行的程序:

- 音频播放器
- Web 服务器

不适合在jail中运行的程序:

- Web 浏览器
- 邮件客户端

Chroot 和 jail的问题

粗粒度策略:

- 全部访问或不访问文件系统(All or nothing access to parts of file system)
- 对某些应用程序不适合, 比如Web浏览器
 - 需要对jail外文件读访问 (比如, 在Gmail中发送附件)

不能阻止一些恶意应用:

- 访问网络
- 尝试crash主机操作系统

大纲

- The Confinement Principle
- 系统调用介入(System Call Interposition)
- 基于虚拟机的隔离
- 软件故障隔离(Software Fault Isolation)

系统调用介入

观察: 为了破坏主机系统, 应用程序必须进行系统调用:

- 删除/覆盖文件: `unlink, open, write`
- 进行网络攻击: `socket, bind, connect, send`

Idea: **监控**应用程序的系统调用并且**阻塞**(block)未授权调用

实现选择:

- 完全在内核中实现 (如, GSWTK, generic software wrapper toolkit)
 - <http://freshmeat.sourceforge.net/projects/gswtk>
- 完全在用户空间中实现 (如, program shepherding)
 - <http://www.burningcutlery.com/derek/docs/security-usenix.pdf>
- 混合 (如, systrace)

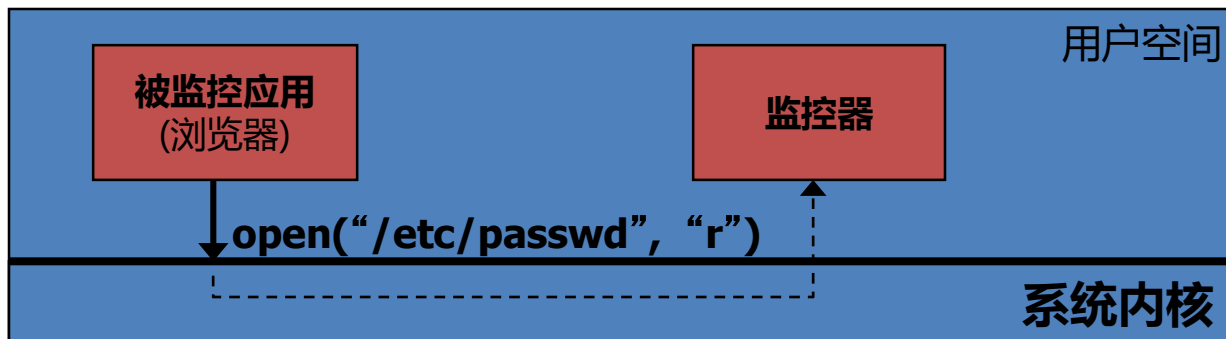
早期实现(Janus)

[GWTB' 96]

Linux **ptrace**: 用于进程追踪

进程调用: **ptrace (... , pid_t pid , ...)**,

当指定pid进程进行系统调用, 唤醒ptrace



如果请求不允许, 监控器kill掉应用程序

实现难点

- 如果应用fork，那么监控器也要fork
 - fork的监控器监控fork的应用程序
- 如果监控器崩溃，应用程序必须被kill掉
- 监控器必须维护与被监控应用程序相关的**所有系统状态**
 - 当前工作目录 (**CWD**), **UID**, **EUID**, **GID**
 - 当应用程序执行 “cd path” 监控器必须更新它的CWD
 - 否则: **相对路径请求解释不正确**

```
cd("/tmp")  
open("passwd", "r")
```

```
cd("/etc")  
open("passwd", "r")
```

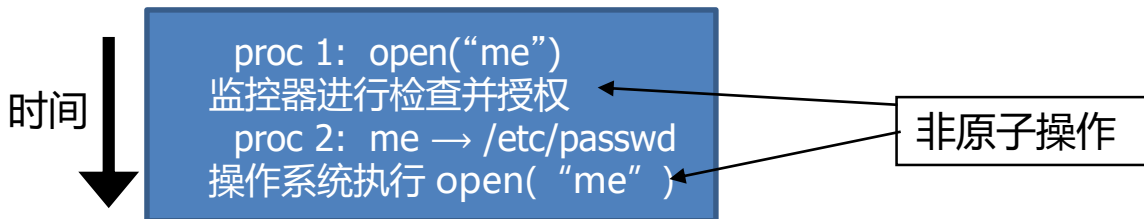
ptrace的问题

ptrace不太适合某些应用程序：

- 需跟踪**所有**系统调用：效率低，如：不需要trace系统调用“close”
- 监控器**只能通过kill**掉应用程序来终止系统调用

安全问题：竞态条件

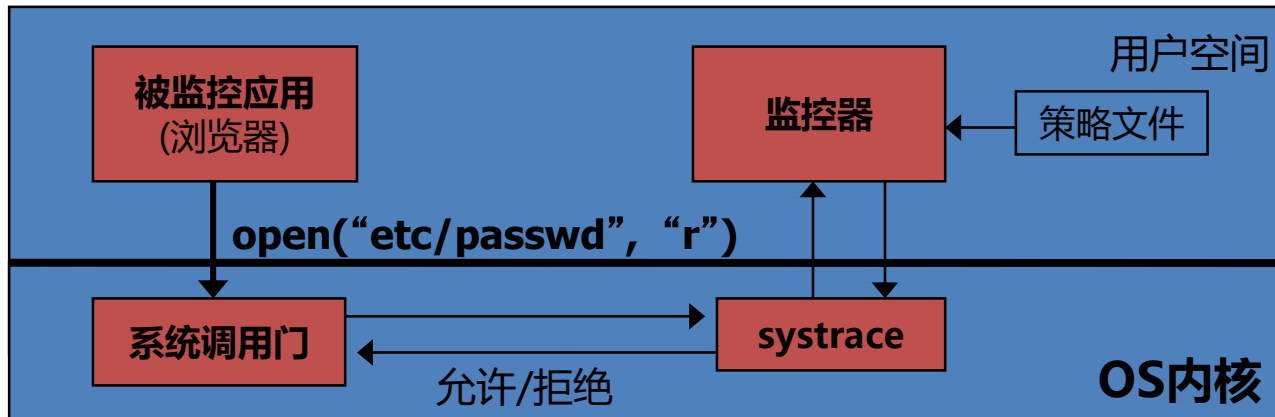
- 例如：建立符号连接：me -> mydata.dat



经典的TOCTOU错误，time-of-check to time-of-use

替代设计: systrace

[P' 02]



- systrace 只转发**被监控的系统调用**给监控器
- systrace 解析符号连接, 并且用**完整目标路径**替代系统调用路径参数
- 当应用程序调用 `execve`, 监控器加载新的过滤策略文件

策略

策略文件示例:

```
path allow /tmp/*  
path deny /etc/passwd  
network deny all
```

手动指定应用的过滤策略比较困难:

- systrace 通过学习应用在“良好”输入上的行为，自动产生策略
- 如果策略未涵盖特定的系统调用，则询问用户
... 但用户也无法决定

另外，为特定应用程序(如：浏览器)选择策略很困难，是这种方法未广泛使用的主要原因

Seccomp安全计算模式

- 2005年, seccomp 出现在 Linux kernel 2.6.12

seccomp 是 "*secure computing mode*" 的缩写。程序进入 seccomp 模式后只能执行 `_exit()`, `sigreturn()`, `read()`, `write()` 四种系统调用, 如果尝试执行其它的系统调用, 程序会被 SIGKILL 信号杀死

```
#include <stdio. h>           //源码
#include <sys/prctl.h>
#include <sys/socket.h >
#include <linux/seccomp.h>
int main (int argc, char* argv[])
{
    printf ("Install seccomp\n");
    prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT);
    printf("Creating socket\n");
    int sock = socket (AF_INET, SOCK_STREAM, 0);
    return 0;
}
```

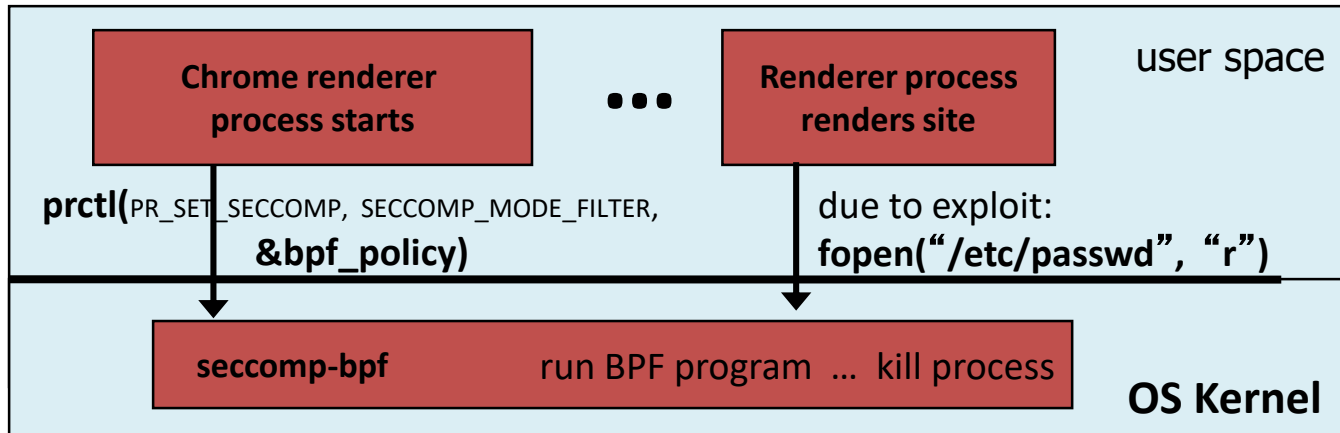
```
$ g++ -o seccomp seccomp.c //运行结果
$ ./seccomp
Install seccomp
Creating socket
Killed
```

```
$ strace ./seccomp //strace 结果
write(1, "Install seccomp\n", Install seccomp)=16
prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT) = 0
write(1, "Creating socket\n", Creating socket)=16
socket (AF_INET, SOCK_STREAM, IPPROTO_IP) = ?
+++ killed by SIGKILL +++
```

seccomp-bpf

seccomp-BPF: Linux **kernel** facility used to **filter** process sys calls

- **sys-call filter** written in the BPF language (use BPFC compiler)
- Used in **Chromium, Docker containers**, ...



BPF filters (policy programs)

Process can install multiple BPF filters:

- once installed, filter cannot be removed (all run on *every syscall*)
 - if program forks, *child inherits all filters*
 - if program calls *execve*, all filters are *preserved*
-

BPF filter input: *syscall number*, *syscall args.*, arch. (x86 or ARM)

Filter returns one of:

- SECCOMP_RET_KILL: kill process
- SECCOMP_RET_ERRNO: return specified error to caller
- SECCOMP_RET_ALLOW: allow syscall

Installing a BPF filter

- Must be called before setting BPF filter.
- Ensures set-UID, set-GID ignored on subsequent execve()
⇒ attacker cannot elevate privilege

```
int main (int argc , char **argv ) {  
    prctl(PR_SET_NO_NEW_PRIVS , 1);  
    prctl(PR_SET_SECCOMP,    SECCOMP_MODE_FILTER,  &bpff_policy);  
    fopen("file.txt",  "w");  
    printf("... will not be printed. \n" );  
}
```

Kill if call open() for write

Example Filters

```
/* Load architecture */
BPF_STMT(BPF_LD | BPF_W | BPF_ABS, (offsetof(struct seccomp_data, arch))),

/* Kill process if the architecture is not what we expect */
BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, AUDIT_ARCH_X86_64, 1, 0),
BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_KILL_PROCESS),

/* Allow system calls other than open() and openat() */
BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, __NR_open, 2, 0),
BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, __NR_openat, 1, 0),
BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW),
BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_KILL_PROCESS)
```

https://man7.org/tlpi/code/online/dist/seccomp/seccomp_deny_open.c

https://man7.org/tlpi/code/online/dist/seccomp/seccomp_control_open.c

<https://eigenstate.org/notes/seccomp.html>

[/usr/include/i386-linux-gnu/asm/unistd_32.h](#)

system call number

```
struct sock_filter {  
    __u16 code;    /* Filter code (opcode)*/  
    __u8  jt;      /* Jump true */  
    __u8  jf;      /* Jump false */  
    __u32 k;       /* Multiuse field (operand) */  
};
```

```
#define BPF_STMT(code, k) \  
    { (unsigned short)(code), 0, 0, k }  
#define BPF_JUMP(code, k, jt, jf) \  
    { (unsigned short)(code), jt, jf, k }
```

https://man7.org/training/download/secisol_seccomp_slides.pdf

Example Filters

Load architecture number into accumulator:

```
BPF_STMT ( BPF_LD | BPF_W | BPF_ABS , (offsetof(struct seccomp_data, arch )))
```

❑ **Opcode** here is constructed by ORing three values together:

✓ BPF_LD: **load**

✓ BPF_W: **operand size** is a word (4 bytes)

✓ BPF_ABS: address mode specifying that source of load is data area (containing system call data)

❑ **Operand** is *architecture* field of data area

✓ `offsetof()` yields byte offset of a field in a structure

Example Filters

Test value in accumulator:

```
BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, AUDIT_ARCH_X86_64, 1, 0)
```

- ▣ BPF_JMP | BPF_JEQ: jump with test on equality
- ▣ BPF_K: value to test against is in generic multiuse field (k)
- ✓ k contains value AUDIT_ARCH_X86_64
- ✓ jt value is 1, meaning **skip one** instruction if test is true
- ✓ jf value is 0, meaning **skip zero** instructions if test is false
 - i.e., continue execution at following instruction

libseccomp

```
int seccomp_init(uint32_t def_action);
int seccomp_rule_add(scmp_filter_ctx ctx, uint32_t action,
    int syscall, unsigned int arg_cnt, ...);
int seccomp_load(void);

seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(close), 0);
seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(write), 1,
    SCMP_A0(SCMP_CMP_EQ, STDERR_FILENO));
```

```

void main(void)
{
    /* initialize the libseccomp context */
    scmp_filter_ctx ctx = seccomp_init(SCMP_ACT_KILL);

    /* allow exiting */
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(exit_group), 0);

    /* allow getting the current pid */
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(getpid), 0);

    /* allow changing data segment size, as required by glibc */
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(brk), 0);

    /* allow writing up to 512 bytes to fd 1 */
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(write), 2, SCMP_A0(SCMP_CMP_EQ, 1),
        SCMP_A2(SCMP_CMP_LE, 512));

    /* if writing to any other fd, return -EBADF */
    seccomp_rule_add(ctx, SCMP_ACT_ERRNO(EBADF), SCMP_SYS(write), 1, SCMP_A0(SCMP_CMP_NE, 1));

    /* load and enforce the filters */
    seccomp_load(ctx);
    seccomp_release(ctx);

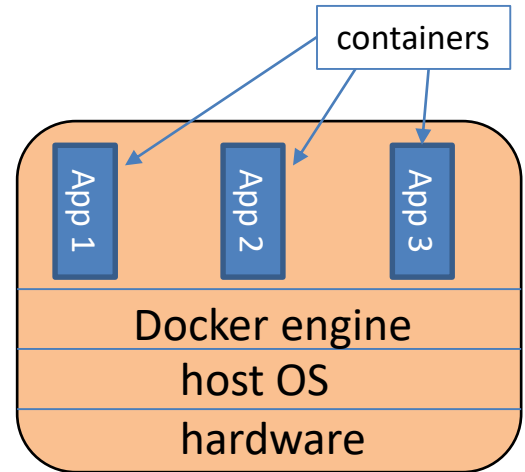
    printf("this process is %d\n", getpid());
}

```

Docker: isolating containers using seccomp-bpf

Container: process level isolation

- Container prevented from making sys calls filtered by seccomp-BPF
- Whoever starts container can specify BPF policy
 - default policy blocks many **syscalls**, including *ptrace*



Docker sys call filtering

Run nginx container with a specific filter called filter.json:

```
$ docker run --security-opt seccomp=filter.json nginx
```

Example filter:

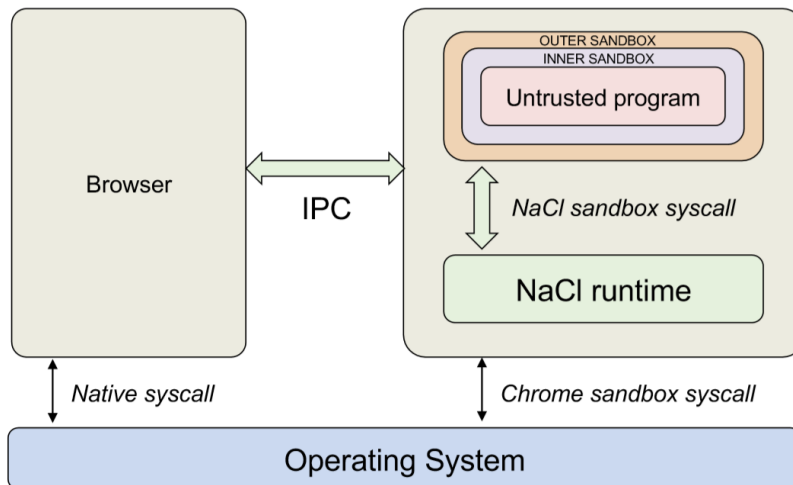
```
"defaultAction": "SCMP_ACT_ERRNO", // deny by default
"syscalls": [
  { "names": ["accept"],           // sys-call name
    "action": "SCMP_ACT_ALLOW",    // allow (whitelist)
    "args": [ ] },                // what args to allow
  ...
]
```

NaCl

- **Google Native Client (NaCl)**

- Native Client是Google在浏览器领域推出的一个开源技术
- 它允许在浏览器内执行**原生的**编译好的代码
- 好处：
 - 为Web提供更多的图形、音频以及其他功能
 - 良好的可移植性
 - 高性能
 - 方便从桌面迁移
 - **高安全性**

NaCl



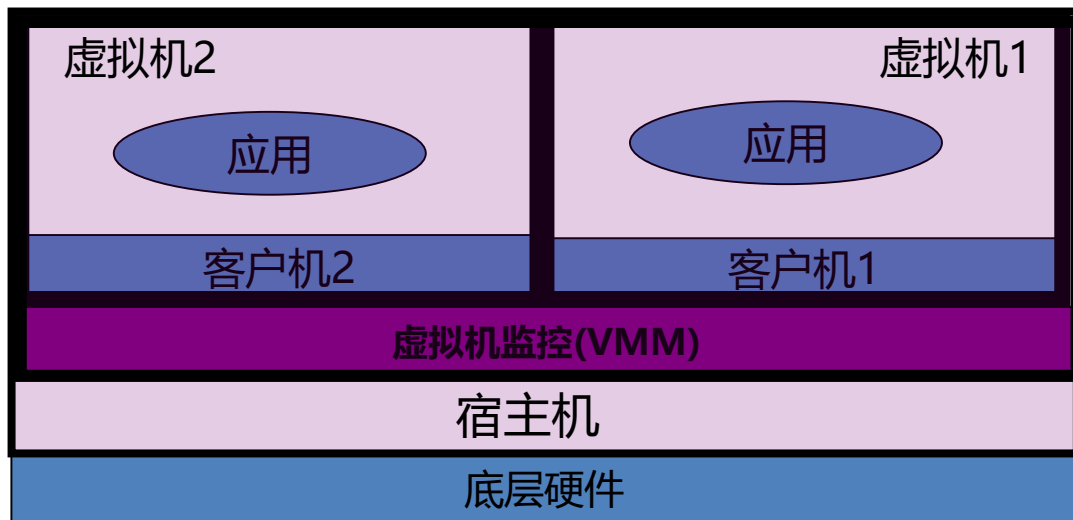
- 不可信代码(如：游戏)
- 两个沙箱：
 - 外部沙箱：使用**系统调用介入**限制功能
 - 内部沙箱：使用 x86 内存分段来隔离不同Web应用内存

Native Client: A Sandbox for Portable, Untrusted x86 Native Code (SP'2012)

大纲

- The Confinement Principle
- 系统调用介入(System Call Interposition)
- 基于虚拟机的隔离
- 软件故障隔离(Software Fault Isolation)

虚拟机



例如： 美国国家安全局NSA的 **NetTop** （迷你型PC）

用于保密和非保密数据处理的单一硬件平台

为什么虚拟机现在很流行？

20世纪60年代：

- 计算机很少，用户很多
- 虚拟机允许多个用户共享单个计算机

20世纪70年代至2000年： 没太大发展

2000至今：

- 计算机太多，用户太少，带来维护负担
 - 打印服务器，邮件服务器，Web 服务器，文件服务器，数据库...
- 计算机性能提高：在不同的硬件上运行每个服务是很浪费的
- 更普遍的：虚拟机在云计算中应用很广

虚拟机监控器安全假设

虚拟机监控器安全假设：

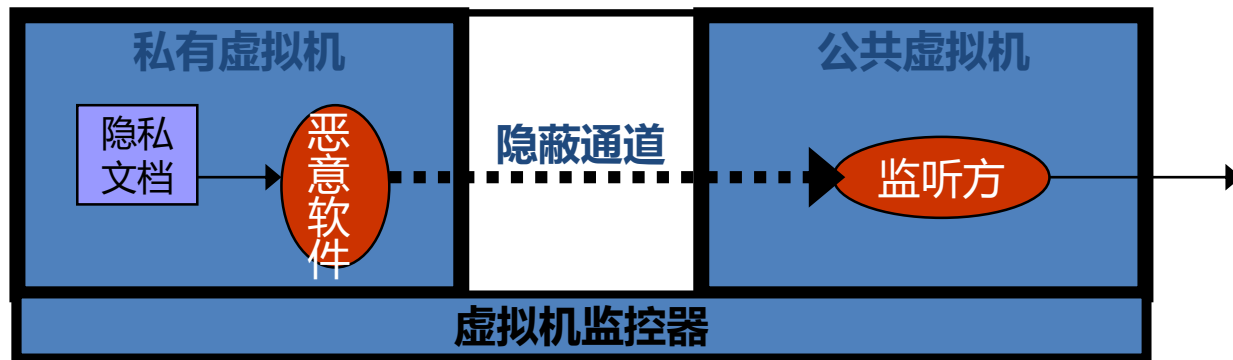
- 恶意软件可能感染客户机操作系统和客户机应用
- 但是恶意软件不能从受感染的虚拟机中逃逸
 - 不能感染**宿主机系统(Host)**
 - 不能感染同一硬件上的**其它虚拟机**

要求虚拟机监控器能保护自身并且没有软件漏洞

- **前提：虚拟机监控器比操作系统简单的多**

问题：隐蔽通道

- **隐蔽通道（Covert Channel）**：隔离组件之间的非预期通信信道
 - 可以用来将机密数据从安全组件泄露到公共组件



隐蔽通道

所有的虚拟机使用相同的底层硬件

发送一个比特位 $b \in \{0,1\}$ 时，恶意软件将：

- $b = 1$ ：在 1:00am 进行CPU 密集型计算
- $b = 0$ ：在 1:00am 不做任何计算

在 1:00am 监听方进行CPU密集型计算，并且测量完成时间

$b = 1 \quad \Leftrightarrow \quad \text{完成时间} > \text{阈值}$

运行中的系统存在大量的隐蔽通道：

- 文件锁状态，缓存，中断 ...
- 很难消除所有的隐蔽通道

假设有问题的系统有两个CPU：私有虚拟机运行在一个CPU上，公共虚拟机运行在另一个CPU上。

虚拟机之间是否有隐蔽通道？

存在隐蔽通道，比如：基于从内存中读取数据所需时间的隐蔽通道

入侵检测/防病毒

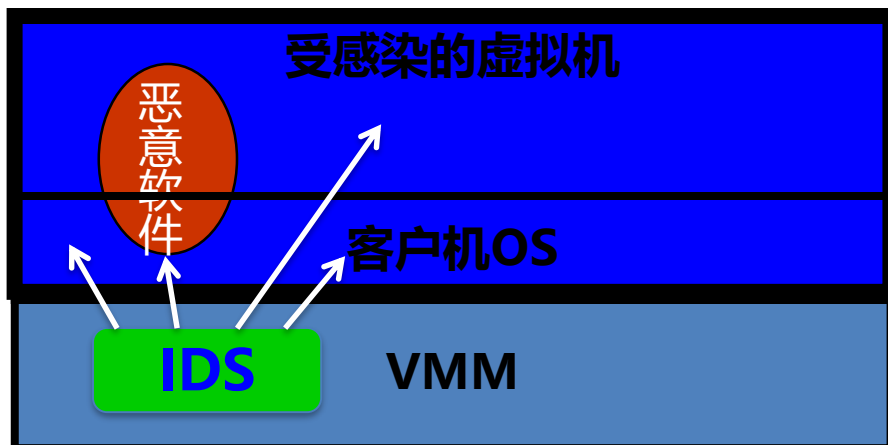
内核rootkit，作为OS内核和用户空间进程的一部分运行：

- 可以关闭系统保护
- 现代恶意软件的常见做法

标准的解决方法：在**网络上**运行入侵检测系统（IDS）

更好的方法：将入侵检测系统**作为VMM的一部分**来运行（**防止恶意软件**）

- VMM 可以监控虚拟硬件的异常情况
- VMI(Virtual Machine Introspection): 虚拟机自省
 - 允许VMM检查客户机OS内部状态



硬件

举例

隐形的rootkit 恶意软件:

- 创建一些 “ps” 命令查不到的进程
- 打开一些 “netstat” 命令查不到的套接字

1. Lie detector检测

- 目的: 检测隐藏进程和网络活动的恶意软件
- 方法:
 - VMM 列出一些运行在客户机中的进程
 - VMM 要求客户机列出进程(如. ps)
 - 如果不匹配: kill 虚拟机

举例

2. 应用代码完整性检测器

- VMM计算客户机中运行的用户应用代码哈希值
- 与哈希表的白名单进行比较
 - 如果有未知应用出现，kill掉虚拟机

3. 确保客户机OS内核的完整性

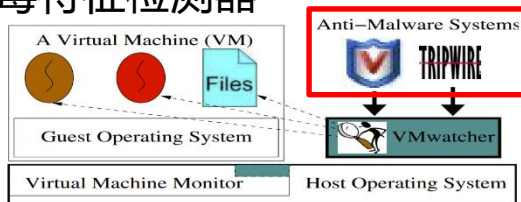
- 例如：检测 `sys_call_table` 是否改变

4. 病毒特征检测器

- 在客户机OS中运行病毒特征检测器



(a) The traditional “in the box” approach



(b) The VMwatcher approach

虚拟监视器VMM恶意利用：Subvirt

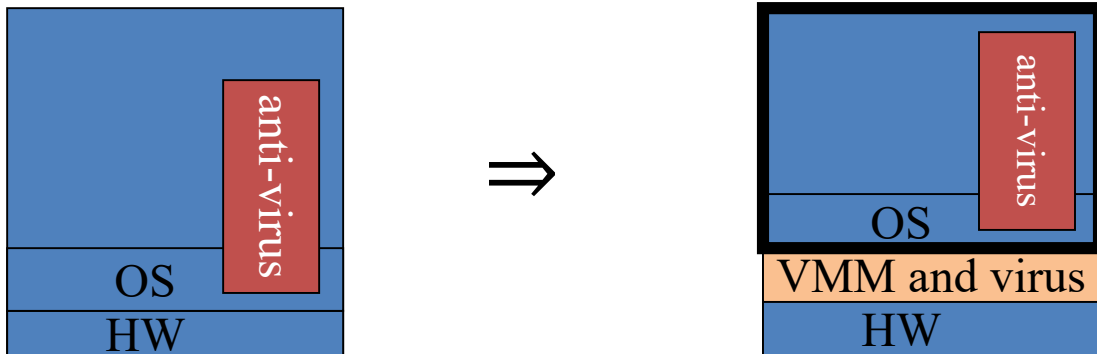
[King et al. 2006]

Idea:

进入Victim计算机后，安装**恶意VMM**

病毒隐藏在VMM中

使得在**VM**内部运行的病毒检测器不可见



VM Based Malware (blue pill virus)

- VMBR (Virtual Machine-based Rootkits) Idea:
 - 利用AMD64 SVM扩展将操作系统移动到虚拟机中（do it 'on-the-fly'）
 - 提供thin hypervisor来控制guest操作系统
 - Hypervisor负责控制guest OS中的事件
- 与依赖VMware或Virtual PC等商业虚拟化技术的SubVirt不同，Blue Pill使用硬件虚拟化并允许操作系统继续直接与硬件通信
- **Blue Pill:** Your operating system swallows the Blue Pill and it awakes inside the Matrix controlled by the **ultra thin Blue Pill hypervisor.**

VMM 检测（需求）

- 操作系统能否检测到它正在 VMM 之上运行？
- 应用：
 - 病毒检测器可以检测VMBR
 - 普通病毒（非VMBR）可以检测VMM
 - 拒绝运行以避免逆向工程
 - 绑定到硬件的软件（例如MS Windows）可以拒绝在VMM上运行
 - DRM系统可能拒绝在VMM之上运行

VMM 检测(Red pill 技术)

- VM平台通常模拟简单的硬件
 - VMWare模拟古老的i440bx芯片组
- VMM引入了时间延迟差异
 - 存在VMM时，内存缓存行为会有所不同
 - 导致任何两个操作的相对时间变化
- VMM与GuestOS共享TLB
 - GuestOS可以检测减少的TLB大小
-以及更多方法[GAWF'07]

VMM 检测

Bottom line: **The perfect VMM does not exist**

- VMMs today (e.g. VMWare) focus on:
 - Compatibility**: ensure **off the shelf software** works
 - Performance**: minimize virtualization **overhead**
- VMMs do not provide **transparency**
 - **Anomalies** reveal existence of VMM

大纲

- The Confinement Principle
- 系统调用介入(System Call Interposition)
- 基于虚拟机的隔离
- 软件故障隔离(Software Fault Isolation)

软件故障隔离(SFI) [Whabe et al., 1993]

目的:

1. 限制运行在相同的地址空间的应用程序

- **编解码代码**不应干扰媒体播放器
- **设备驱动程序**不应破坏内核
- **插件**不应该破坏Web浏览器

2. 允许**有效的跨域调用**

- 媒体播放器和编解码器之间
- 内核和设备驱动之间

简单的解决方法: 在隔离的地址空间中运行App

- **问题: 如果Apps频繁通信, 应用将会很慢**
 - 对于每个消息都要上下文切换

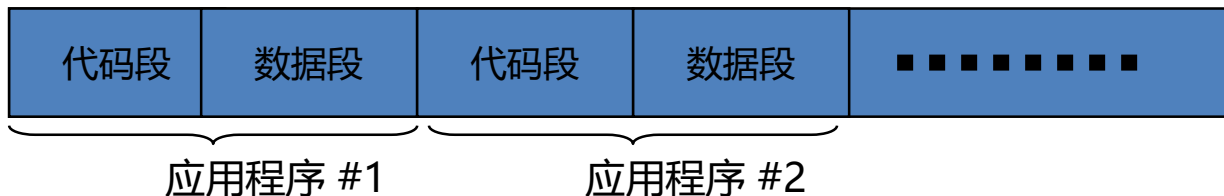
软件故障隔离: 主要思想

- 给每个不可信程序分配一个 **fault domain** (连续的内存空间 + 唯一标识符)
- 修改 不可信程序, **防止** 其写入或者跳转到 **fault domain** 以外的地址

软件故障隔离

SFI 方法:

- 将进程地址划分成不同的段



- **定位不安全指令: jmp, load, store**
 - 在编译时, 在不安全指令前添加**防护模块**(guards)
 - 当加载代码时, 确保所有的防护模块都在

软件故障隔离: In More Detail

- 代码分为代码和数据段
 - 跳转目标仅限于代码段
 - 数据地址仅限于数据段
- 代码和数据段地址存储在专用寄存器中
- 在执行任何引用内存的指令之前
 - 将内存引用的段地址与存储的段地址（寄存器中）进行比较
 - 如果地址不相等，则停止执行
- 应用
 - User-level file systems, Google's NativeClient

段匹配技术

- **dr1, dr2**: 未被程序执行使用的专用寄存器
 - **dr2 包含段ID**

- 间接 load 指令 **R12 \leftarrow [R34]** 变成:

```
dr1  $\leftarrow$  R34
scratch-reg  $\leftarrow$  (dr1 >> 20)
compare scratch-reg and dr2
trap if not equal
R12  $\leftarrow$  [dr1]
```

防护模块确保代码不从另一段加载数据

: 获取段ID

: 验证段ID

: 执行 load

地址沙箱技术

- **dr2**: 包含段 ID
- 间接 load 指令 **R12 ← [R34]** 变成:

dr1 ← R34 & segment-mask

dr1 ← dr1 | dr2

R12 ← [dr1]

: 将段ID位清零

: 设置有效的段ID

: 执行 load

- 与段匹配技术相比, 需要较少的指令
... **但是不能截获恶意指令**

Registers

- 问题
 - Dedicated Register
 - Scratch Register
- CISC架构下， registers不够使用！

Guard zones

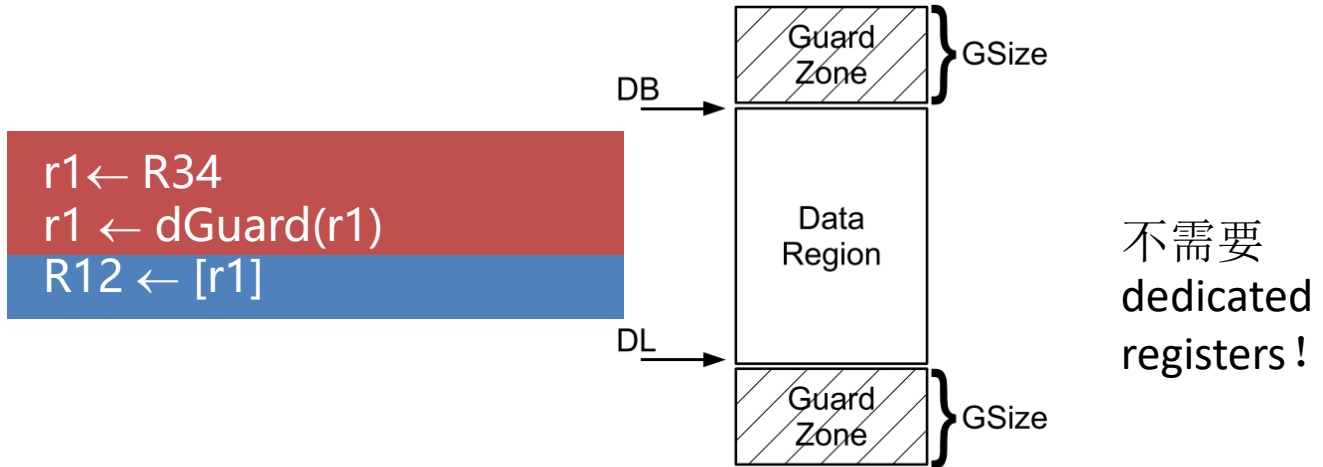


Figure 3.3: Data region surrounded by guard zones (from Zeng *et al.*, 2011).

NaCl-x86-64 (Sehr et al., 2010)在4GB sandbox的上下，使用40GB大小的guard zone

Scratch Registers

- Binary level
 - Binary-level IRM rewriting
 - Liveness analysis: 利用dead register，作为scratch register
 - Stack: 借助stack保存registers
- Compiler level
 - Reserve dedicated scratch registers inside the compiler
 - 举例: SFI toolchain of PittSField (McCamant and Morrisett, 2006) reserves *ebx* as the scratch register
 - Rewriting at the level of a compiler IR (e.g., the LLVM IR)

问题: 如果 `jmp [addr]` 直接跳转到间接load指令, 怎么办?
(绕过防护模块)

解决方案:

`jmp` 防护必须保证 `[addr]` 不会绕过 load的防护模块

Indirect-jump control-flow enforcement

CISC架构

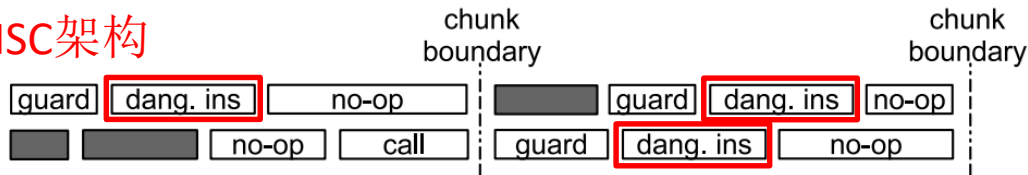


Figure 3.5: Illustration of the aligned-chunk enforcement. Black-filled rectangles represent regular (non-dangerous) instructions. Rectangles with “dang. ins” represent dangerous instructions, which are preceded by guards. For alignment, no-op instructions have to be inserted. Furthermore, call instructions are placed at the end of chunks since return addresses must be aligned.

```
r := r & 0x1000FFF0
```

```
jmp r
```

```
[0x10000000,0x1000FFFF]
```

Since a **guard** and its **guarded instruction** are in **one chunk**,
and **jumps** target **only the beginnings of chunks**,
the guard cannot be bypassed by jumps.

MIPS架构，使用专用register，register只能被monitor code修改，应用程序不能修改

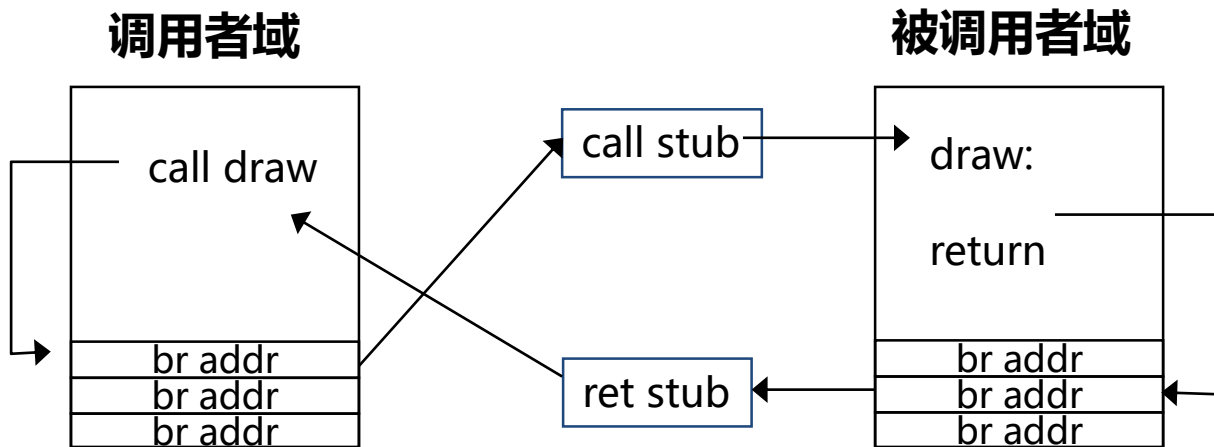
Control-Flow Integrity in NaCl

- For each **direct branch**, statically compute target and verify that it's a valid instruction
 - Must be reachable by fall-through disassembly
- **Indirect branches** must be encoded as

```
and %eax, 0xffffffe0
jmp *%eax
```

 - Guarantees that target is **32-byte aligned**
 - Very efficient enforcement of control-flow integrity
- No RET
 - Sandboxing sequence, then **indirect jump**

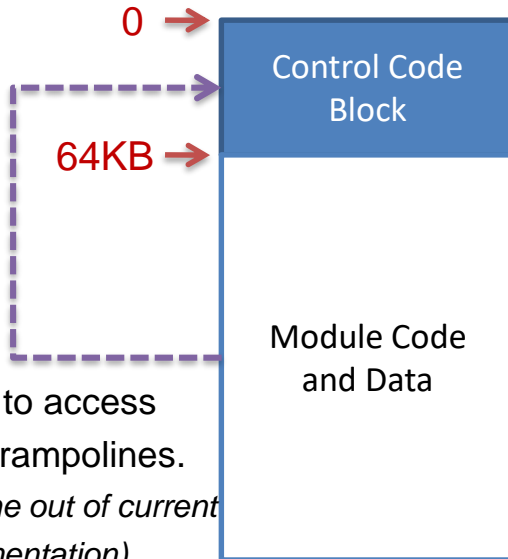
跨域调用



- 只有 **stubs** 被允许执行跨域跳转
- 跳转表包含允许的退出点
 - 地址是**硬编码**的，所在的段只读

Load a NaCl module

Memory address:



User **far call** to access
system call trampolines.
(call the routine out of current
memory segmentation)

All **far calls** are under
control

1. Verify the **module code** according to the obligations.
2. Load **control code block** into memory (*including system call trampolines, thread context data*).
3. Load the module code and data into memory.
4. Set the **segment registers** to establish a private memory space (*64KB afterwards, 64KB is the zero offset*).
5. **Transfer** the control to the module code.

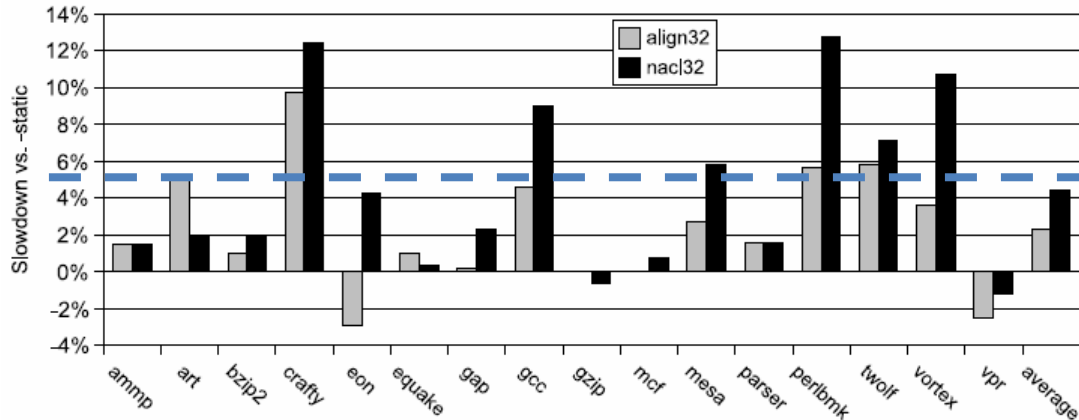
性能

Benchmark		DEC-MIPS					DEC-ALPHA	
		Fault Isolation Overhead	Protection Overhead	Reserved Register Overhead	Instruction Count Overhead	Fault Isolation Overhead (predicted)	Fault Isolation Overhead	Protection Overhead
052.alvinn	FP	1.4%	33.4%	-0.3%	19.4%	0.2%	8.1%	35.5%
bps	FP	5.6%	15.5%	-0.1%	8.9%	5.7%	4.7%	20.3%
cholesky	FP	0.0%	22.7%	0.5%	6.5%	-1.5%	0.0%	9.3%
026.compress	INT	3.3%	13.3%	0.0%	10.9%	4.4%	-4.3%	0.0%
056.ear	FP	-1.2%	19.1%	0.2%	12.4%	2.2%	3.7%	18.3%
023.eqntott	INT	2.9%	34.4%	1.0%	2.7%	2.2%	2.3%	17.4%
008.espresso	INT	12.4%	27.0%	-1.6%	11.8%	10.5%	13.3%	33.6%
001.gcc1.35	INT	3.1%	18.7%	-9.4%	17.0%	8.9%	NA	NA
022.li	INT	5.1%	23.4%	0.3%	14.9%	11.4%	5.4%	16.2%
locus	INT	8.7%	30.4%	4.3%	10.3%	8.6%	4.3%	8.7%
mp3d	FP	10.7%	10.7%	0.0%	13.3%	8.7%	0.0%	6.7%
psgrind	INT	10.4%	19.5%	1.3%	12.1%	9.9%	8.0%	36.0%
qcd	FP	0.5%	27.0%	2.0%	8.8%	1.2%	-0.8%	12.1%
072.sc	INT	5.6%	11.2%	7.0%	8.0%	3.8%	NA	NA
tracker	INT	-0.8%	10.5%	0.4%	3.9%	2.1%	10.9%	19.9%
water	FP	0.7%	7.4%	0.3%	6.7%	1.5%	4.3%	12.3%
Average		4.3%	21.8%	0.4%	10.5%	5.0%	4.3%	17.6%

store 和 jump 检查

load, store 和 jump 检查

性能-NaCl



Max space overhead is **57.5%** code size increment for *gcc* in SPEC CPU 2000.

Mandatory alignment for jump targets impacts the instruction cache and increases the code size (*more **significant** if compared to **dynamic linked** executables*).

软件故障隔离总结

- **共享内存:** 使用虚拟内存硬件
 - 在地址空间内, 映射相同的物理页到两个段中
- **性能**
 - 一般很好: mpeg文件播放, 4%性能下降
- **软件故障隔离的局限性:** 较难在x86平台上实现:
 - 变长指令: 不知道什么地方放置防护模块
 - 寄存器很少: 无法满足SFI至少需要3个专用寄存器的需求
 - NaCl使用x86段寄存器
 - 许多影响内存的指令: 需要更多的防护

总结

- **多种沙箱技术:**

- 物理隔离, 虚拟化隔离(VMMs),

- 系统调用介入, 软件故障隔离

- 专用应用(如. 浏览器中的javascript沙箱)

- **经常完全隔离是不合适的**

- Apps往往需要通过常规接口通信

- **沙箱技术的难点:**

- 指定策略: app可以做什么, 不可以做什么

- 防止隐蔽通道

思考

- 一、理解并实践Freebsd jail或者Jailkit方法；
- 二、理解Chrome中的隔离和约束是如何实现的？
- 三、查找相关参考文献，理解基于虚拟化的云系统中，隐蔽通道是如何实现的？

FreeBSD Jail，通常简称为jail，是一种轻量级的虚拟化技术，它允许管理员在单一的FreeBSD系统上隔离运行多个独立的系统实例，即“jail”。每个jail看起来和操作起来都像一个完整的Unix系统，但实际运行在一个受限的环境中，只能访问特定的资源和文件系统。实践FreeBSD jail的方法包括配置jail环境、设置网络接口、分配资源限制以及管理jail的启动和停止。Jail提供了一种有效的隔离手段，可用于测试新软件、运行不受信任的应用程序或创建虚拟主机环境。

Jailkit是一种类似于FreeBSD jail的技术，但它适用于Linux系统。Jailkit允许管理员创建受限的用户环境，用户在其中拥有一定程度的系统访问权限，但受限于特定的资源和应用程序。实践Jailkit的方法包括安装Jailkit工具、配置用户权限、设置chroot环境以及管理用户的资源访问。

Chrome浏览器的隔离和约束是通过多个层面实现的。首先，Chrome使用沙箱技术将不同的进程和插件隔离开来，以减少恶意软件对系统的影响。其次，Chrome的多进程架构确保了浏览器的各个组件，如渲染进程、扩展进程等，都在独立的进程中运行，彼此之间通过IPC（进程间通信）进行数据交换。此外，Chrome还实施了严格的内容安全策略（CSP），限制网页可以执行的操作，以及强化了对恶意网站的检测和过滤。

在基于虚拟化的云系统中，隐蔽通道的实现通常涉及到虚拟机之间的通信，这种通信可能不被云平台的管理工具所检测到。例如，攻击者可能会利用虚拟机共享的物理资源，如内存或CPU，来进行隐蔽的数据传输。这可能通过分析资源使用模式、共享的文件系统或网络接口来实现。此外，云环境中的隐蔽通道也可能通过虚拟化软件的漏洞或配置不当来建立。