

# CPU 虚拟化

学号： N/A

姓名： N/A

专业： 计算机科学与技术

# 1 调研部分

## 1.1 可虚拟化架构

在系统级虚拟化中，每台虚拟机中都有属于它的虚拟硬件，虚拟机监控器 Hypervisor 最核心的部分是处理器虚拟化，它负责将虚拟机的指令翻译成真实硬件的指令。如果物理机和虚拟机使用的指令集相同，为了提高虚拟机性能，虚拟机的大部分指令可以在处理器上直接运行，Hypervisor 可以只模拟操作敏感物理资源的指令，这种指令称为敏感指令。

现代指令集架构通常都有两个或两个以上的特权级，用来分隔系统软件和应用软件。系统中的管理关键系统资源的指令称为特权指令。如果在非最高特权级中执行这些指令将会触发异常中断，陷入最高特权级中。

在可虚拟化架构中，所有的敏感指令都是特权指令。将 Hypervisor 运行在系统的最高特权级上，那么所有的敏感指令都会陷入最高特权级由 Hypervisor 模拟。这样，Hypervisor 就可以保证虚拟机的正常运行，同时保护物理机的安全。

相反，如果有些敏感指令不是特权指令，那么 Hypervisor 就无法通过特权级切换捕获这些指令，产生虚拟化漏洞。这些指令集架构不能原生支持虚拟化，只能通过软件模拟来实现虚拟化，性能会受到影响。这样的架构称为不可虚拟化架构。

一个架构是否可虚拟化，取决于架构中的敏感指令和特权指令的关系。如果一个架构中敏感指令都是特权指令，那么这个架构就是可虚拟化架构，反之则是不可虚拟化架构。

## 1.2 陷入再模拟

“陷入再模拟”（trap-and-emulate）是一种虚拟化技术，用于在虚拟机中执行特权指令和敏感指令。它的基本思想是，当虚拟机执行特权指令时，会触发一个异常中断，然后由虚拟机监控器 Hypervisor 来模拟 emulate 该指令的行为。

“陷入再模拟”出现的前提条件是虚拟机的非特权指令可以直接在硬件上执行，而特权指令和敏感指令需要在 Hypervisor 中模拟。这样，虚拟机可以在硬件级别运行，而 Hypervisor 只需要在虚拟机执行特权指

令时介入。如果指令集不同，虚拟机的所有指令都需要在 Hypervisor 中模拟，不会出现“陷入再模拟”的情况。

“陷入再模拟”还需要指令集是可虚拟化的，即敏感指令都是特权指令。如果敏感指令不是特权指令，那么 Hypervisor 无法通过特权级切换捕获这些指令，会出现虚拟机逃逸漏洞。

“陷入再模拟”技术可以提高虚拟机的性能，因为大部分指令可以在硬件上直接运行，只有少部分指令需要在 Hypervisor 中模拟。这样，不需要对应用程序修改，也不需要额外的硬件支持，就可以使虚拟机的性能接近于物理机。

### 1.3 Intel VT-x 的特权级划分

Intel VT-x 是 Intel 的虚拟化技术，它引入了两个新的特权级：VMX root 和 VMX non-root。VMX root 是处理器的最高特权级，用来运行 Hypervisor。VMX non-root 是处理器的非最高特权级，用来运行虚拟机。

在新的特权级中，处理器引入了 VMCS (Virtual Machine Control Structure) 结构，用来保存虚拟机的状态。在 VMX root 中，Hypervisor 可以通过 VMCS 来控制虚拟机的运行。

同时，处理器引入了 VMXON 和 VMXOFF 指令，用来在 VMX root 和 VMX non-root 之间切换。在切换时，处理器会保存当前的状态到 VMCS 中，然后加载新的 VMCS 中的状态。这样，Hypervisor 可以通过 VMXON 和 VMXOFF 指令来 CPU 的虚拟化状态。

在 VMX non-root 中，处理器可以在硬件级别判断虚拟机是否执行了敏感指令，如果执行了敏感指令，处理器会陷入 VMX root，由 Hypervisor 模拟这些指令。这种方式不仅解决了虚拟化漏洞，还极大地提高了虚拟机的性能。

## 2 实验目的

- 理解实验通过 ioctl 调用 KVM 创建虚拟机的流程，理解设置虚拟 CPU、虚拟内存、虚拟机寄存器内容的方法。
- 实现虚拟机执行 I/O 指令时的“陷入再模拟”机制，将虚拟机向端口写入的字符串转换为小写字符串输出。

## 3 实验步骤

### 3.1 实验环境

- 操作系统: Ubuntu 22.04.3 LTS
- 内核版本: 5.15.0-86-generic x86\_64
- 编译环境: gcc version 11.4.0
- 编辑环境: Visual Studio Code

### 3.2 实验过程

#### 3.2.1 连接实验环境

通过 VSCode Remote 使用 SSH 连接到实验环境。

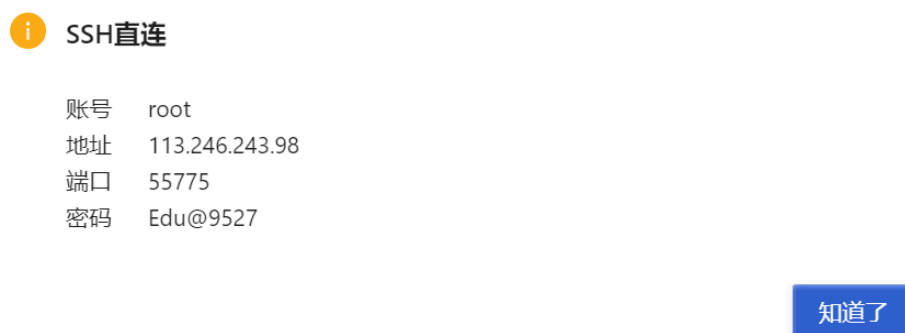


Figure 1: 查看实验环境登录信息

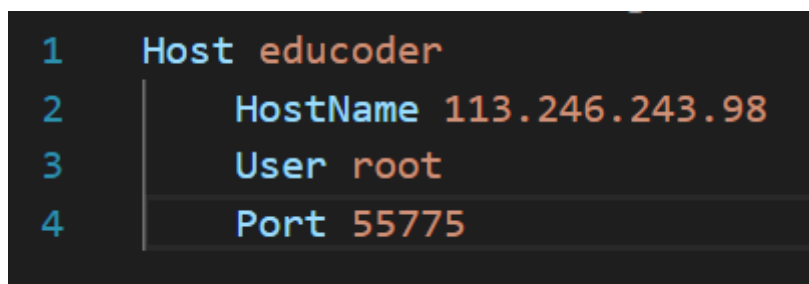


Figure 2: SSH 连接到实验环境

使用 `lsb_release -a`、`uname -a`、`gcc --version` 等命令查看实验环境信息。

```
● root@virtlab:/# lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 22.04.3 LTS
Release:        22.04
Codename:       jammy
● root@virtlab:/# uname -r
5.15.0-86-generic
● root@virtlab:/# gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/11/lto-wrapper
OFFLOAD_TARGET_NAMES=nvptx-none:amdgcn-amdhsa
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 11.4.0-1ubuntu1' --with-tune=generic --enable-checking=release --build=x86_64-linux-gnu --disable-multilib --without-cuda-driver --enable-offload-targets=nvptx-none=/build/gcc-11-X/n/usr --without-cuda-driver --enable-checking=release --build=x86_64-linux-gn
k-serialization=2
Thread model: posix
Supported LTO compression algorithms: zlib zstd
gcc version 11.4.0 (Ubuntu 11.4.0-1ubuntu1~22.04)
○ root@virtlab:/#
```

Figure 3: 查看系统信息

进入/home/virtlab/labs/cpu lab 目录，查看实验文件。

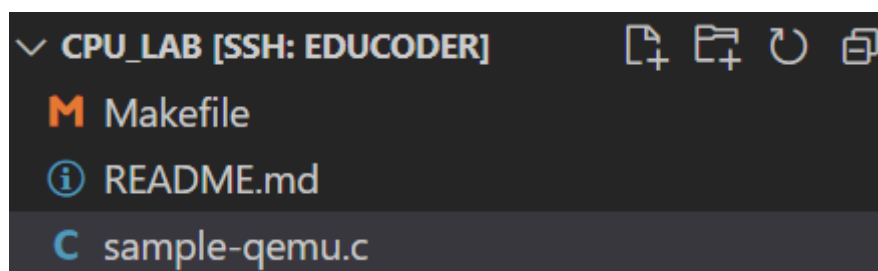


Figure 4: 查看实验文件

### 3.2.2 理解虚拟机的创建过程

- 实验代码首先使用 open 函数打开 /dev/kvm 设备文件，以便与 KVM 交互。如果打开失败，程序会报错并退出。

```
kvm = open("/dev/kvm", O_RDWR | O_CLOEXEC);
if (kvm == -1)
    err(1, "/dev/kvm");
```

- 使用 `ioctl` 函数调用 `KVM_GET_API_VERSION` 命令，获取 KVM API 版本。如果版本不匹配，程序也会报错并退出。

```
ret = ioctl(kvm, KVM_GET_API_VERSION, NULL);
if (ret == -1)
    err(1, "KVM_GET_API_VERSION");
if (ret != 12)
    errx(1, "KVM_GET_API_VERSION %d, expected 12", ret);
```

- 使用 `ioctl` 函数创建一个新的虚拟机，返回的文件描述符 `vmfd` 用于后续的虚拟机操作。

```
vmfd = ioctl(kvm, KVM_CREATE_VM, (unsigned long)0);
if (vmfd == -1)
    err(1, "KVM_CREATE_VM");
```

- 使用 `mmap` 函数分配一页对齐的内存来存放虚拟机代码，并使用 `memcpy` 将汇编代码复制到分配的内存中。

```
mem = mmap(NULL, 0x1000, PROT_READ | PROT_WRITE, MAP_SHARED |
MAP_ANONYMOUS, -1, 0);
if (!mem)
    err(1, "allocating guest memory");
memcpy(mem, code, sizeof(code));
```

- 使用 `ioctl` 函数设置虚拟机内存区域，将分配的内存映射到虚拟机的物理地址空间。

```
struct kvm_userspace_memory_region region = {
    .slot          = 0,
    .guest_phys_addr = 0x1000,
    .memory_size    = 0x1000,
    .userspace_addr  = (uint64_t)mem,
};
ret = ioctl(vmfd, KVM_SET_USER_MEMORY_REGION, &region);
if (ret == -1)
    err(1, "KVM_SET_USER_MEMORY_REGION");
```

- 使用 `ioctl` 函数创建一个新的虚拟 CPU，返回的文件描述符 `vcpufd` 用于后续的虚拟 CPU 操作。

```
vcpufd = ioctl(vmfd, KVM_CREATE_VCPU, (unsigned long)0);
if (vcpufd == -1)
    err(1, "KVM_CREATE_VCPU");
```

- 使用 `ioctl` 函数获取虚拟 CPU 的映射大小，并使用 `mmap` 函数映射虚拟 CPU 的内存区域到用户空间。

```
ret = ioctl(kvm, KVM_GET_VCPU_MMAP_SIZE, NULL);
if (ret == -1)
    err(1, "KVM_GET_VCPU_MMAP_SIZE");
mmap_size = ret;
if (mmap_size < sizeof(*run))
    errx(1, "KVM_GET_VCPU_MMAP_SIZE unexpectedly small");
run = mmap(NULL, mmap_size, PROT_READ | PROT_WRITE,
MAP_SHARED, vcpufd, 0);
if (!run)
    err(1, "mmap vcpu");
```

- 使用 `ioctl` 函数获取和设置段寄存器，初始化 CS 段寄存器指向 0，RIP 寄存器指向虚拟机代码的起始地址 0x1000。同时也设置了 RAX、RBX、RFLAGS 等寄存器的初始值。

```
ret = ioctl(vcpufd, KVM_GET_SREGS, &sregs);
if (ret == -1)
    err(1, "KVM_GET_SREGS");
sregs.cs.base = 0;
sregs.cs.selector = 0;
ret = ioctl(vcpufd, KVM_SET_SREGS, &sregs);
if (ret == -1)
    err(1, "KVM_SET_SREGS");
```

```
struct kvm_regs regs = {
    .rip = 0x1000,
    .rax = 2,
    .rbx = 2,
    .rflags = 0x2,
};
ret = ioctl(vcpufd, KVM_SET_REGS, &regs);
if (ret == -1)
    err(1, "KVM_SET_REGS");
```

- 使用 `ioctl` 函数循环运行虚拟机，并在陷入时处理虚拟机退出原因。

```
while (1)
{
```

```

    ret = ioctl(vcpufd, KVM_RUN, NULL);
    if (ret == -1)
        err(1, "KVM_RUN");

    // KVM trap-and-emulate mechanism
    switch (run->exit_reason) {...}
}

```

### 3.2.3 实现虚拟机执行 I/O 指令时的“陷入再模拟”机制

- 在虚拟机运行时，如果遇到 KVM\_EXIT\_IO 退出原因，说明虚拟机执行了 I/O 指令，需要模拟 I/O 操作。虚拟机通过结构体 kvm\_run 的 io 字段获取 I/O 操作的信息。io 字段包含的信息通常如下：
  - direction: I/O 操作方向，使用 KVM\_EXIT\_IO\_OUT 表示输出操作。
  - size: I/O 操作大小。
  - port: I/O 操作端口。
  - count: I/O 操作计数。
  - data\_offset: I/O 操作数据相对于 kvm\_run 结构体的偏移量。
- 根据实验要求，当 I/O 操作满足以下所有条件时，将 I/O 写入的字符转换为小写字符输出，否则报错退出。
  - I/O 操作方向为 KVM\_EXIT\_IO\_OUT；
  - I/O 操作大小为 1 字节；
  - I/O 操作次数为 1；
  - I/O 操作端口为 0x3f8。

```

case KVM_EXIT_IO:
    if (run->io.direction != KVM_EXIT_IO_OUT)
        errx(...);
    if (run->io.size != 1)
        errx(...);
    if (run->io.count != 1)
        errx(...);
    if (run->io.port != 0x3f8)
        errx(...);

    // Print the character
    some_print_code();

```

- 通过 run 的 data\_offset 字段获取 I/O 写入的字符相对于 run 结构体的偏移量，然后将字符转换为小写字符输出。

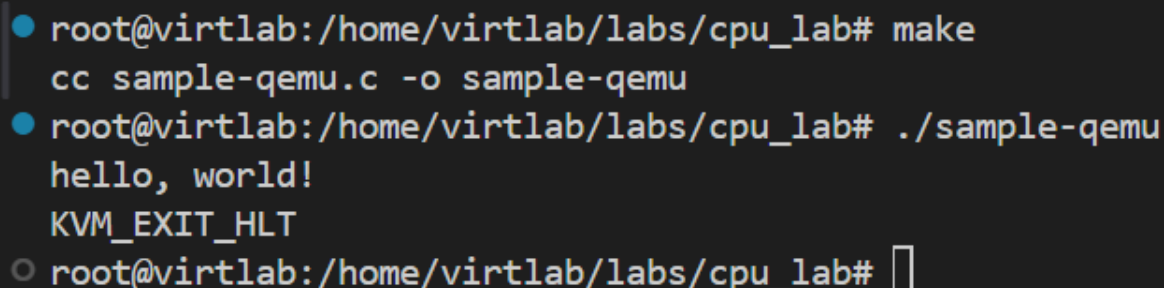


```
char *data_addr = (char *)(run) + run->io.data_offset;  
putchar(tolower(*data_addr));
```

### 3.3 实验结果

使用 make 命令编译实验代码，生成可执行文件 sample-qumu。

运行 ./sample-qumu 命令，输出如图所示：



```
● root@virtlab:/home/virtlab/labs/cpu_lab# make  
cc sample-qemu.c -o sample-qemu  
● root@virtlab:/home/virtlab/labs/cpu_lab# ./sample-qemu  
hello, world!  
KVM_EXIT_HLT  
○ root@virtlab:/home/virtlab/labs/cpu_lab#
```

Figure 5: 实验结果

## 4 实验分析

在实验中，我们通过 ioctl 调用 KVM API 创建了一个虚拟机，并实现了虚拟机执行 I/O 指令时的“陷入再模拟”机制。当虚拟机执行预定代码时，遇到 I/O 指令和 hlt 指令时，虚拟机会陷入再模拟，届时可以使用 ioctl 函数获取虚拟机的退出原因，并根据退出原因进行相应的处理。

在实验中，我们通过 KVM\_EXIT\_IO 退出原因获取 I/O 操作的信息，在 I/O 导致的虚拟机陷入时，将 I/O 写入的字符转换为小写字符输出。实验结果表明，虚拟机成功执行了预定代码，并正确输出了小写字符串。

随后虚拟机执行了 hlt 指令，虚拟机退出原因为 KVM\_EXIT\_HLT，程序正常退出。输出符合预期，实验结果正确。

## 附录

### 参考文献

- 教材 深入浅出系统虚拟化：原理与实践
- KVM API Documentation: <https://www.kernel.org/doc/Documentation/virtual/kvm/api.txt>

## 程序源码

```

/* Sample code for /dev/kvm API
 *
 * Copyright (c) 2015 Intel Corporation
 * Author: Josh Triplett <josh@joshtriplett.org>
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
copy
 * of this software and associated documentation files (the "Software"), to
 * deal in the Software without restriction, including without limitation the
 * rights to use, copy, modify, merge, publish, distribute, sublicense, and/or
 * sell copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS
 * IN THE SOFTWARE.
 */

#include <ctype.h>
#include <err.h>
#include <fcntl.h>
#include <linux/kvm.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>

int main(void)
{
    int          kvm, vmfd, vcpufd, ret;
    const uint8_t code[] = {
        0xba, 0xf8, 0x03, // mov $0x3f8, %dx
        0xb0, 'H',        // mov $'H', %al
        0xee,              // out %al, (%dx)
        0xb0, 'e',        // mov $'e', %al
        0xee,              // out %al, (%dx)
        0xb0, 'l',        // mov $'l', %al
        0xee,              // out %al, (%dx)
    };

```

```

    0xb0, 'l',          // mov $'l', %al
    0xee,              // out %al, (%dx)
    0xb0, 'o',          // mov $'o', %al
    0xee,              // out %al, (%dx)
    0xb0, ',',          // mov $',', %al
    0xee,              // out %al, (%dx)
    0xb0, ' ',          // mov $' ', %al
    0xee,              // out %al, (%dx)
    0xb0, 'w',          // mov $'w', %al
    0xee,              // out %al, (%dx)
    0xb0, 'o',          // mov $'o', %al
    0xee,              // out %al, (%dx)
    0xb0, 'r',          // mov $'r', %al
    0xee,              // out %al, (%dx)
    0xb0, 'l',          // mov $'l', %al
    0xee,              // out %al, (%dx)
    0xb0, 'd',          // mov $'d', %al
    0xee,              // out %al, (%dx)
    0xb0, '!',          // mov $'!', %al
    0xee,              // out %al, (%dx)
    0xb0, '\n',         // mov $'\n', %al
    0xee,              // out %al, (%dx)
    0xf4,              // hlt
};

uint8_t      *mem;
struct kvm_sregs sregs;
size_t       mmap_size;
struct kvm_run *run;

kvm = open("/dev/kvm", O_RDWR | O_CLOEXEC);
if (kvm == -1)
    err(1, "/dev/kvm");

/* Make sure we have the stable version of the API */
ret = ioctl(kvm, KVM_GET_API_VERSION, NULL);
if (ret == -1)
    err(1, "KVM_GET_API_VERSION");
if (ret != 12)
    errx(1, "KVM_GET_API_VERSION %d, expected 12", ret);

vmfd = ioctl(kvm, KVM_CREATE_VM, (unsigned long)0);
if (vmfd == -1)
    err(1, "KVM_CREATE_VM");

/* Allocate one aligned page of guest memory to hold the code. */
mem = mmap(NULL, 0x1000, PROT_READ | PROT_WRITE, MAP_SHARED |
MAP_ANONYMOUS, -1, 0);
if (!mem)
    err(1, "allocating guest memory");
memcpy(mem, code, sizeof(code));

/* Map it to the second page frame (to avoid the real-mode IDT at 0). */

```

```

struct kvm_userspace_memory_region region = {
    .slot          = 0,
    .guest_phys_addr = 0x1000,
    .memory_size    = 0x1000,
    .userspace_addr  = (uint64_t)mem,
};
ret = ioctl(vmfd, KVM_SET_USER_MEMORY_REGION, &region);
if (ret == -1)
    err(1, "KVM_SET_USER_MEMORY_REGION");

vcpufd = ioctl(vmfd, KVM_CREATE_VCPU, (unsigned long)0);
if (vcpufd == -1)
    err(1, "KVM_CREATE_VCPU");

/* Map the shared kvm_run structure and following data. */
ret = ioctl(kvm, KVM_GET_VCPU_MMAP_SIZE, NULL);
if (ret == -1)
    err(1, "KVM_GET_VCPU_MMAP_SIZE");
mmap_size = ret;
if (mmap_size < sizeof(*run))
    errx(1, "KVM_GET_VCPU_MMAP_SIZE unexpectedly small");
run = mmap(NULL, mmap_size, PROT_READ | PROT_WRITE, MAP_SHARED, vcpufd,
0);
if (!run)
    err(1, "mmap vcpu");

/* Initialize CS to point at 0, via a read-modify-write of sregs. */
ret = ioctl(vcpufd, KVM_GET_SREGS, &sregs);
if (ret == -1)
    err(1, "KVM_GET_SREGS");
sregs.cs.base = 0;
sregs.cs.selector = 0;
ret = ioctl(vcpufd, KVM_SET_SREGS, &sregs);
if (ret == -1)
    err(1, "KVM_SET_SREGS");

/* Initialize registers: instruction pointer for our code, addends, and
initial flags required by x86 architecture. */
struct kvm_regs regs = {
    .rip = 0x1000,
    .rax = 2,
    .rbx = 2,
    .rflags = 0x2,
};
ret = ioctl(vcpufd, KVM_SET_REGS, &regs);
if (ret == -1)
    err(1, "KVM_SET_REGS");

/* Repeatedly run code and handle VM exits. */
while (1)
{
    ret = ioctl(vcpufd, KVM_RUN, NULL);

```

```

    if (ret == -1)
        err(1, "KVM_RUN");
    switch (run->exit_reason)
    {
    case KVM_EXIT_HLT: puts("KVM_EXIT_HLT"); return 0;
    case KVM_EXIT_IO:
        /* Check support conditions */
        if (run->io.direction != KVM_EXIT_IO_OUT)
            errx(1, "unhandled KVM_EXIT_IO direction %d", run->io.direction);

        if (run->io.size != 1)
            errx(1, "unhandled KVM_EXIT_IO size %d", run->io.size);

        if (run->io.count != 1)
            errx(1, "unhandled KVM_EXIT_IO count %d", run->io.count);

        if (run->io.port != 0x3f8)
            errx(1, "unhandled KVM_EXIT_IO port 0x%x", run->io.port);

        /* Print the character */
        char *data_addr = (char *) (run) + run->io.data_offset;
        putchar(tolower(*data_addr));
        break;
    case KVM_EXIT_FAIL_ENTRY:
        errx(1, "KVM_EXIT_FAIL_ENTRY: hardware_entry_failure_reason = 0x%llx",
            (unsigned long long) run->fail_entry.hardware_entry_failure_reason);
    case KVM_EXIT_INTERNAL_ERROR: errx(1, "KVM_EXIT_INTERNAL_ERROR: suberror = 0x%x", run->internal.suberror);
    default:
        errx(1, "exit_reason = 0x%x", run->exit_reason);
    }
}
}

```