School of Computer Science & Engineering

Trustworthy Systems Group

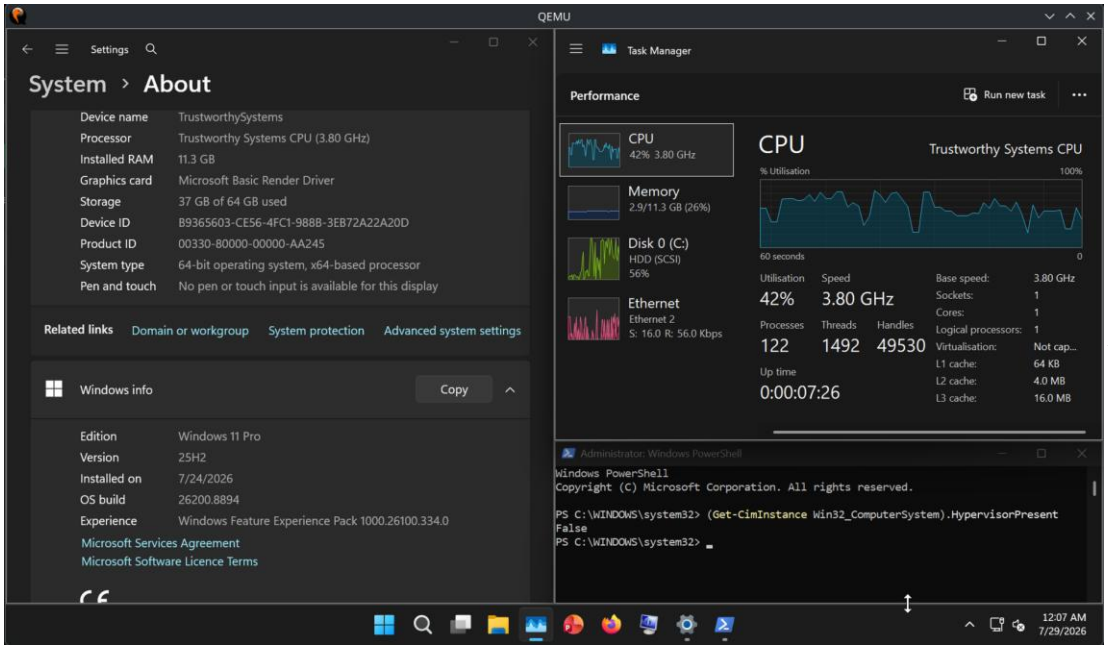
Booting Windows on an x86-64 seL4-based VMM

Bill Nguyen

bill.nguyen@unsw.edu.au

Windows 11. On libvmm.

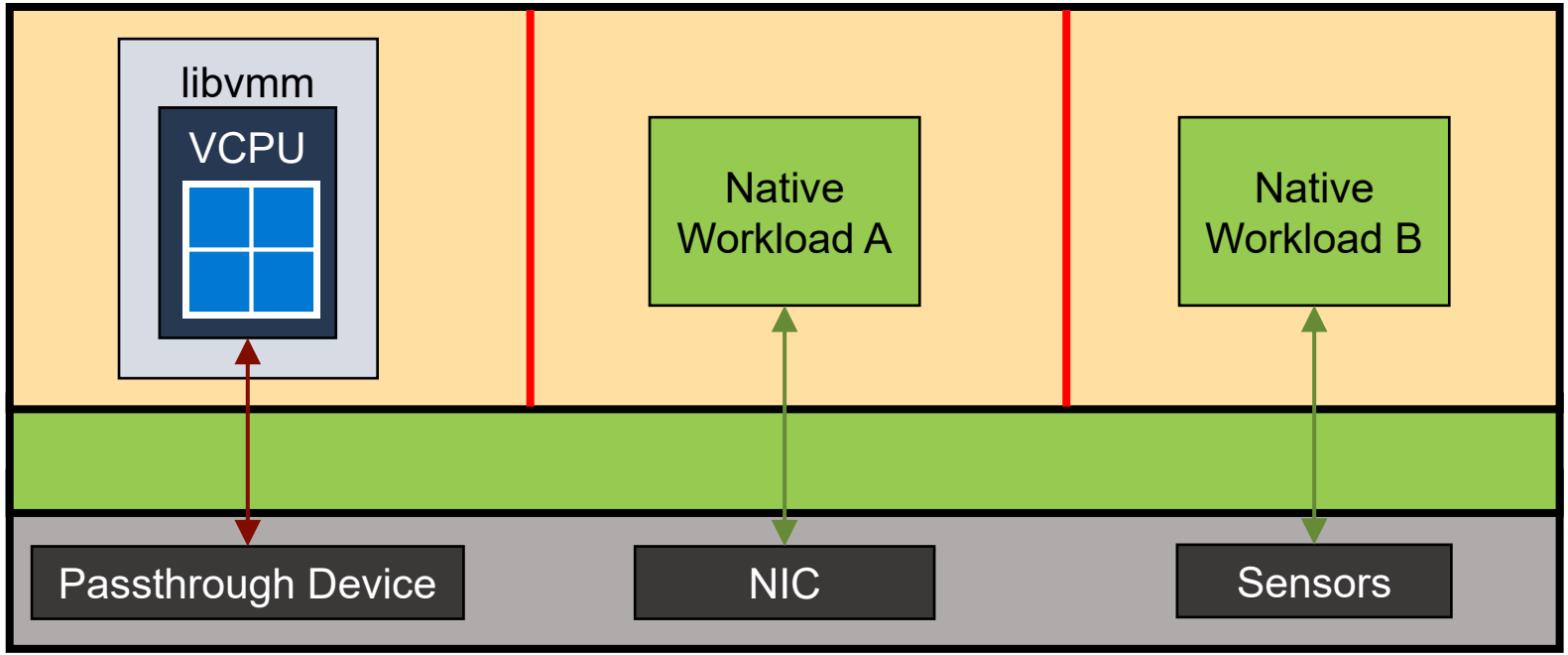
- libvmm: library for making VMMs on seL4 Microkit.
- Unmodified Windows 11 Pro deployment from retail install media on libvmm:



Live Demo at the end.

Why?

- Mixed-criticality systems that need both legacy Windows workload and seL4's guarantees.



seL4 Microkit

Prior Art

	libvmm	CAmkES VMM
OS Framework	Microkit	CAmkES
Target architectures	ARM + x86	ARM + x86
Drivers	Integration with sDDF	Uses its own drivers
Guest boot mechanism	Direct Linux kernel load, or OVMF UEFI firmware	Direct kernel load only
Guest Support	Windows 10, 11 (x86 only) and Linux	Linux
Performance	Hardware accelerated APIC virtualisation (Intel APICv)	Software emulated APIC

Booting Linux

- Linux hands you a documented direct boot protocol.

```
memcpy(kernel_dest,     kernel_blob, kernel_size);
memcpy(initrd_dest,     initrd_blob, initrd_size);
memcpy(cmdline_dest,    cmdline_str, cmdline_size);
memcpy(zero_page_dest, zero_page,    zero_page_size);
memcpy(acpi_dest,       acpi_tables, tables_size);

/* Set up kernel page table */
/* Resume VCPU at Linux handover offset in long mode. */
```

Booting other OSes

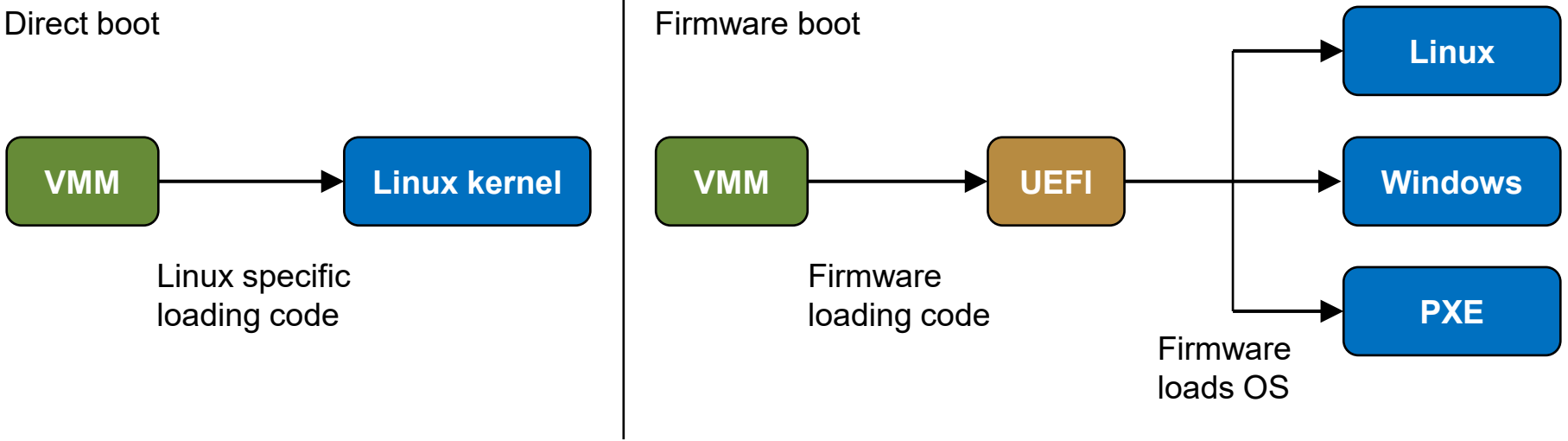
- Windows/GRUB/Limine hand you an EFI executable that expects UEFI services.

```
/* Mount EFI partition */
/* Read in Windows Boot Manager */

memcpy(somewhere??, bootmgfw_efi, size);

/* Resume VCPU at ??? */
```

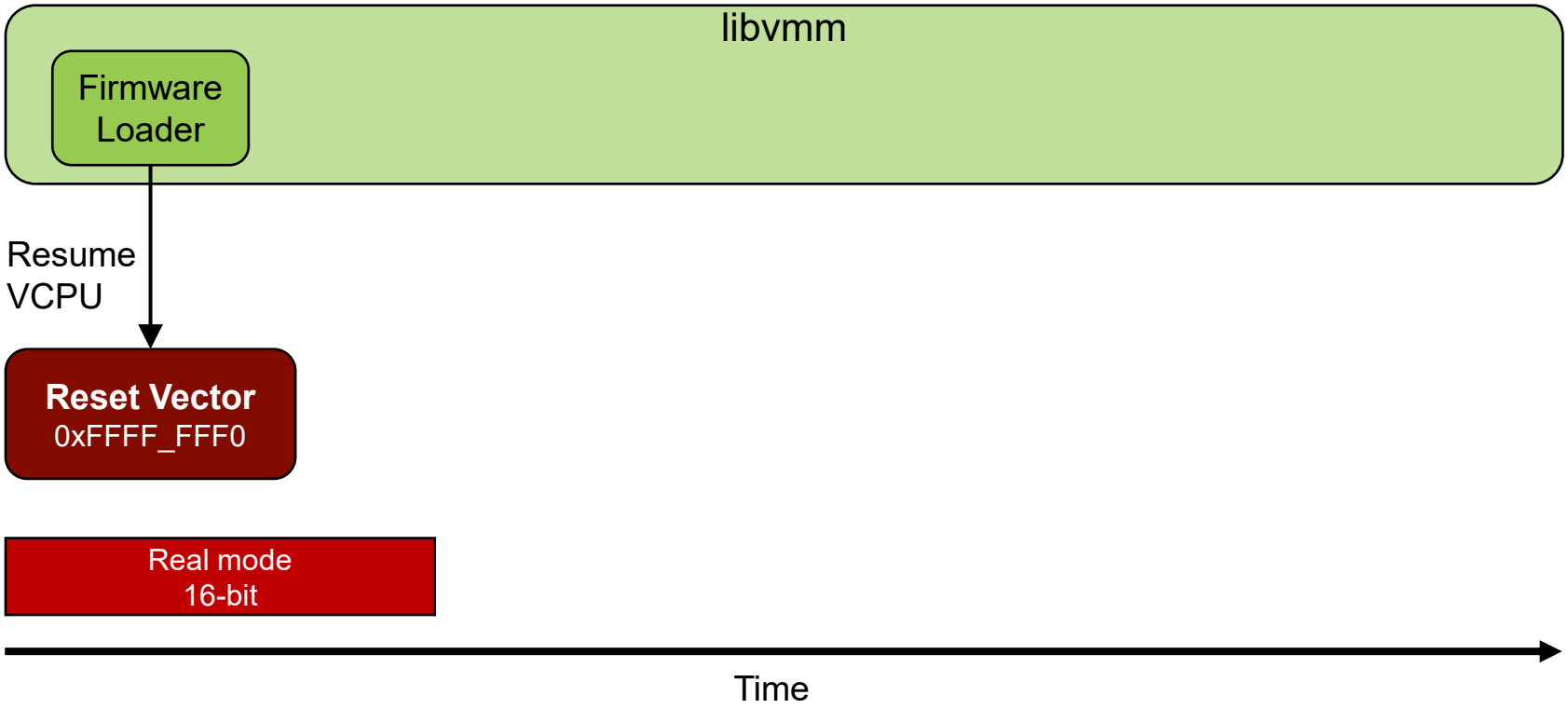
- No direct boot, this won't work.

Booting other OSes

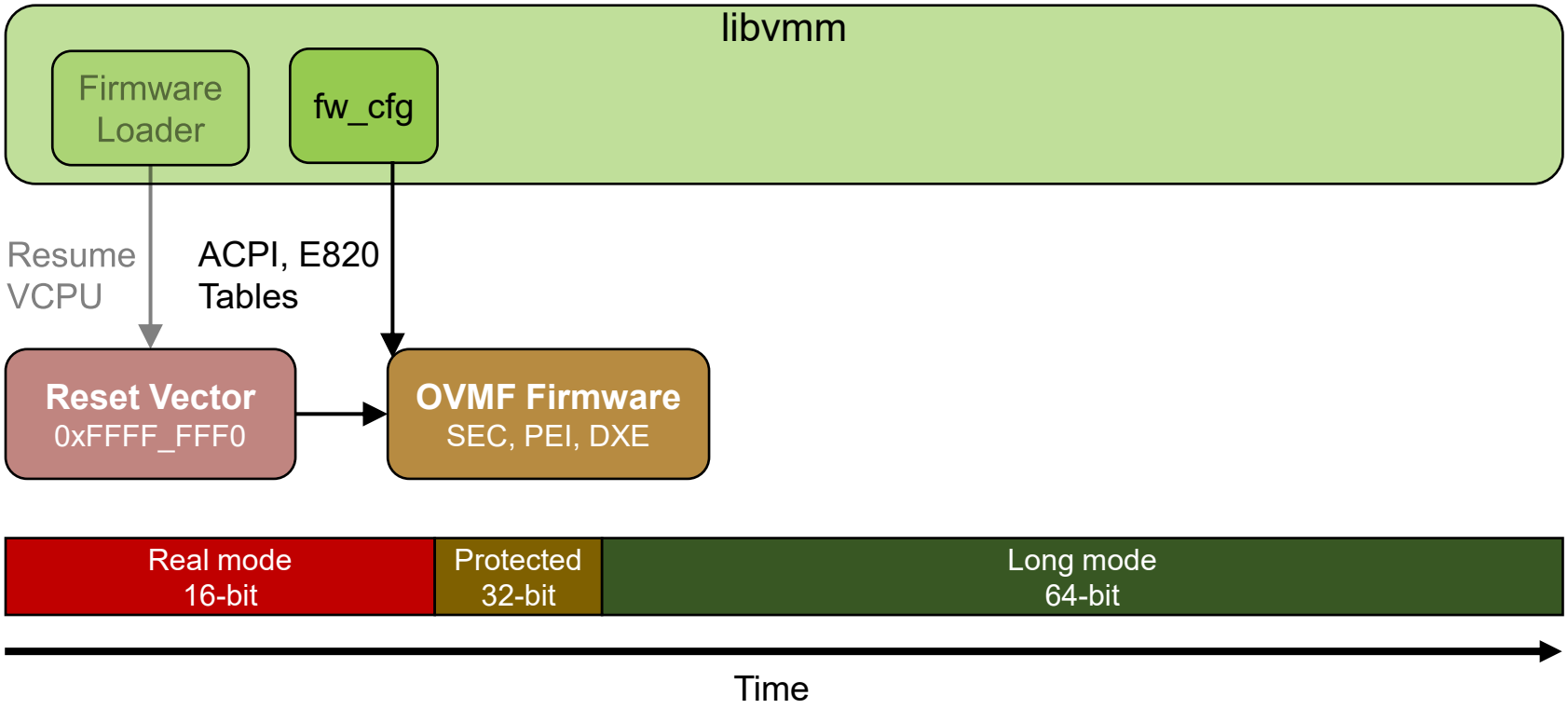
- The guest boot path needs to be generalised.
- Load OVMF UEFI, then let the firmware load *any* guest OS.
- Enabled by seL4 PR #1600 “x86/vcpu: Support guest-managed mode transitions and Unrestricted Guest”, landed in seL4 15.0.0.



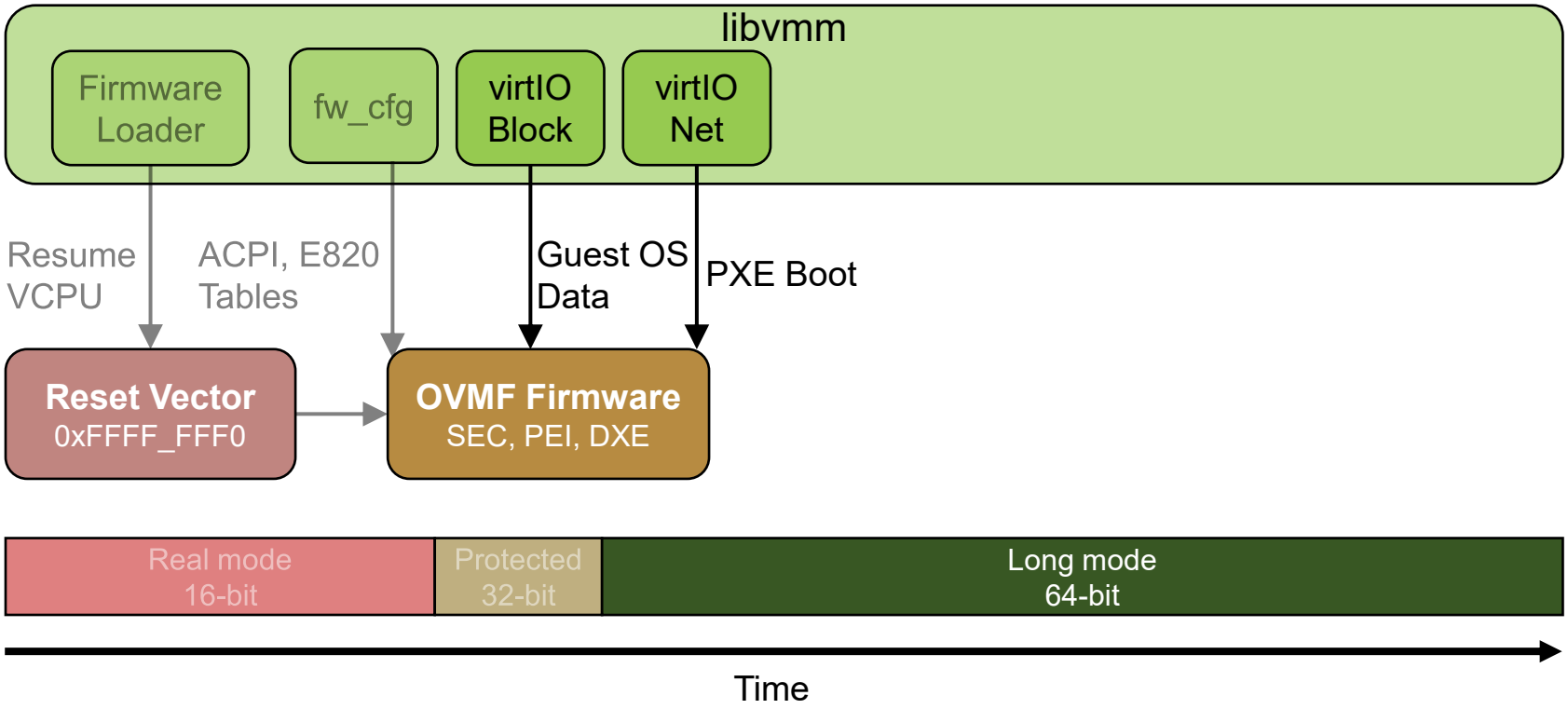
Booting other OSes



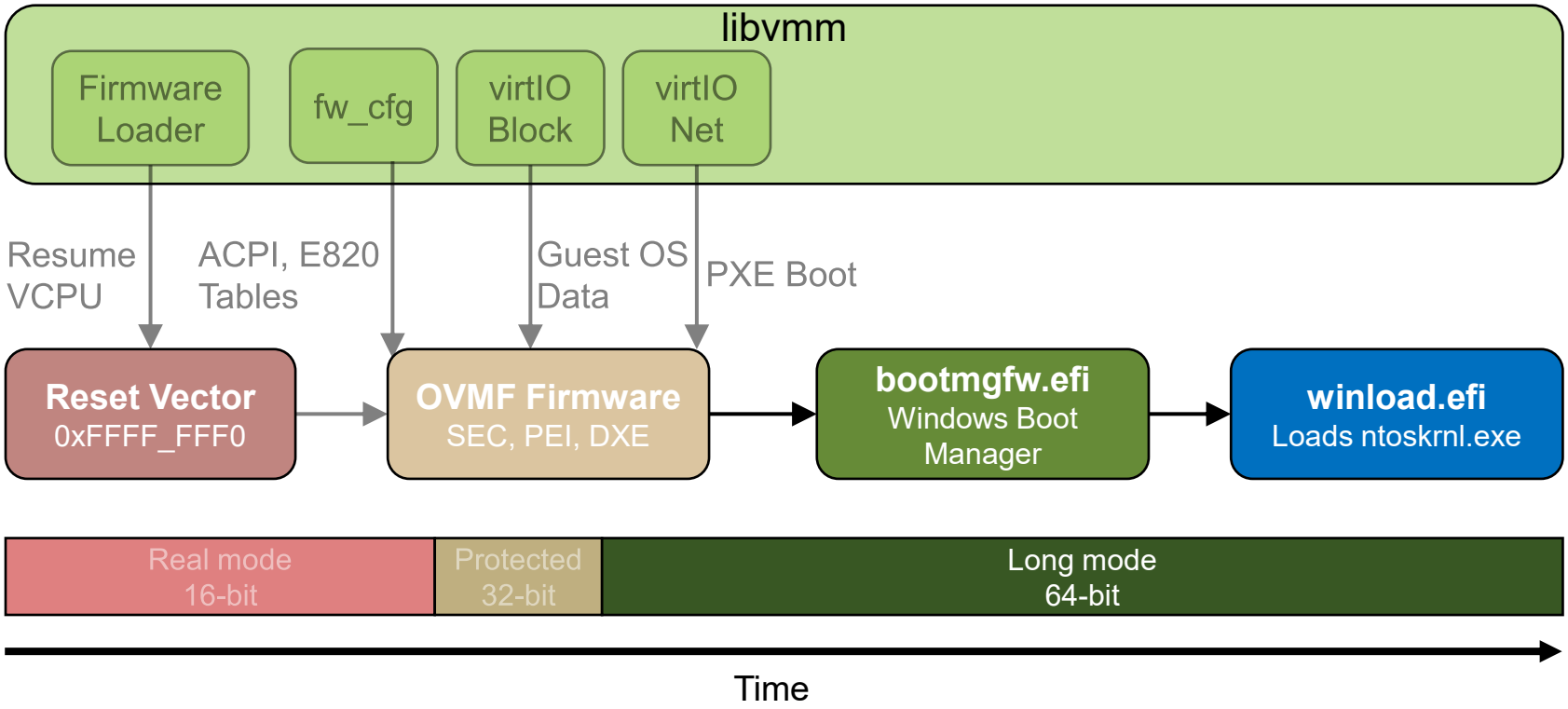
Booting other OSes



Booting other OSes



Booting other OSes

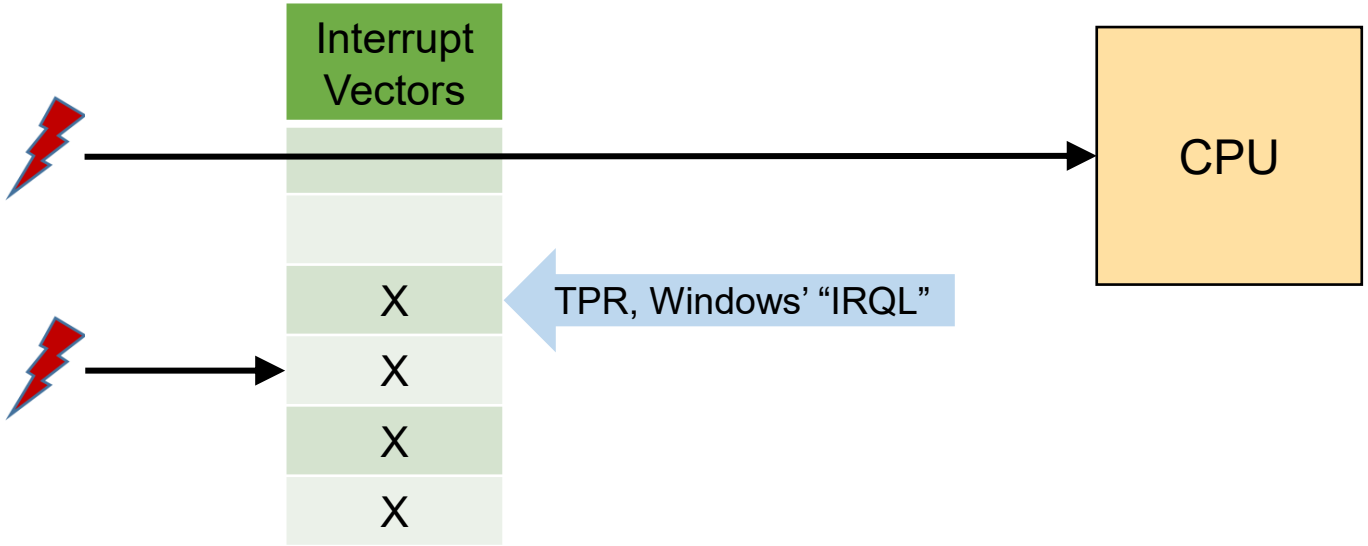


Intel APICv

- Once UEFI worked, Windows booted.
- It took >10 minutes to load an unusably slow desktop. Why?

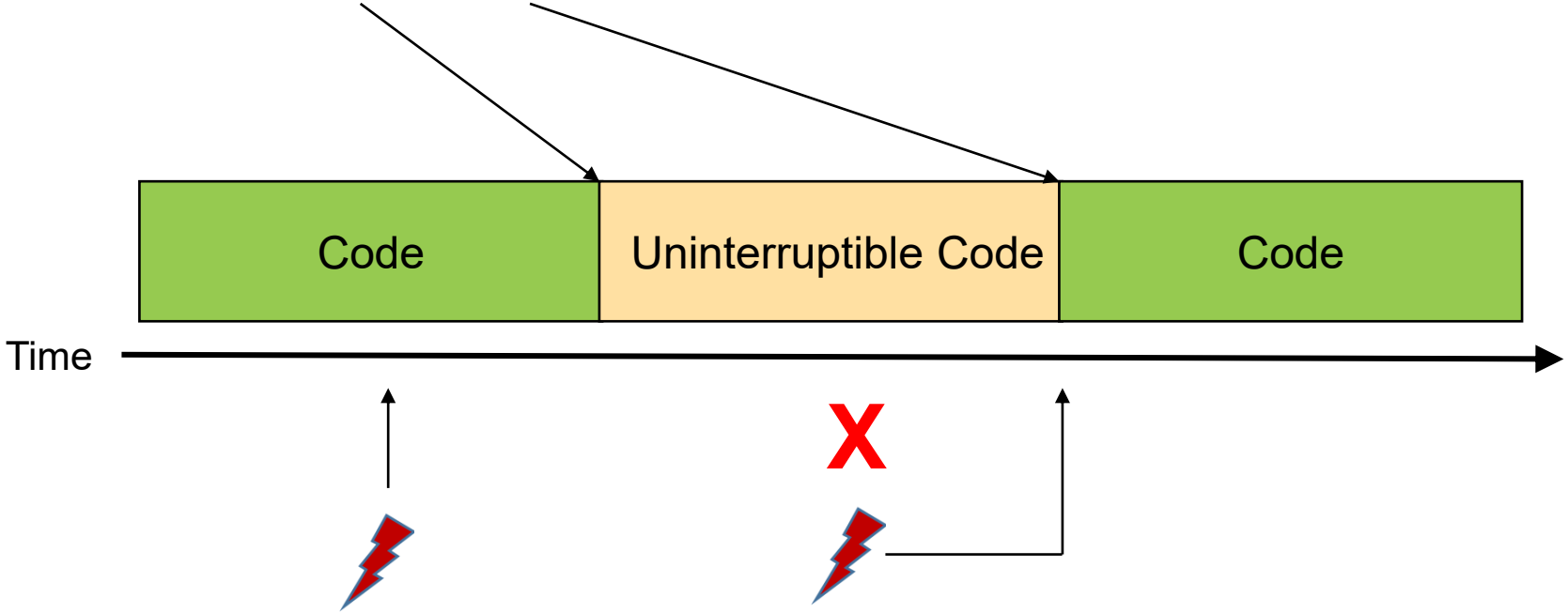
Intel APICv

- Once UEFI worked, Windows booted.
- It took >10 minutes to load an unusably slow desktop. Why?
- Windows reprograms the APIC's Task Priority Register (TPR) 50,000 times a second!



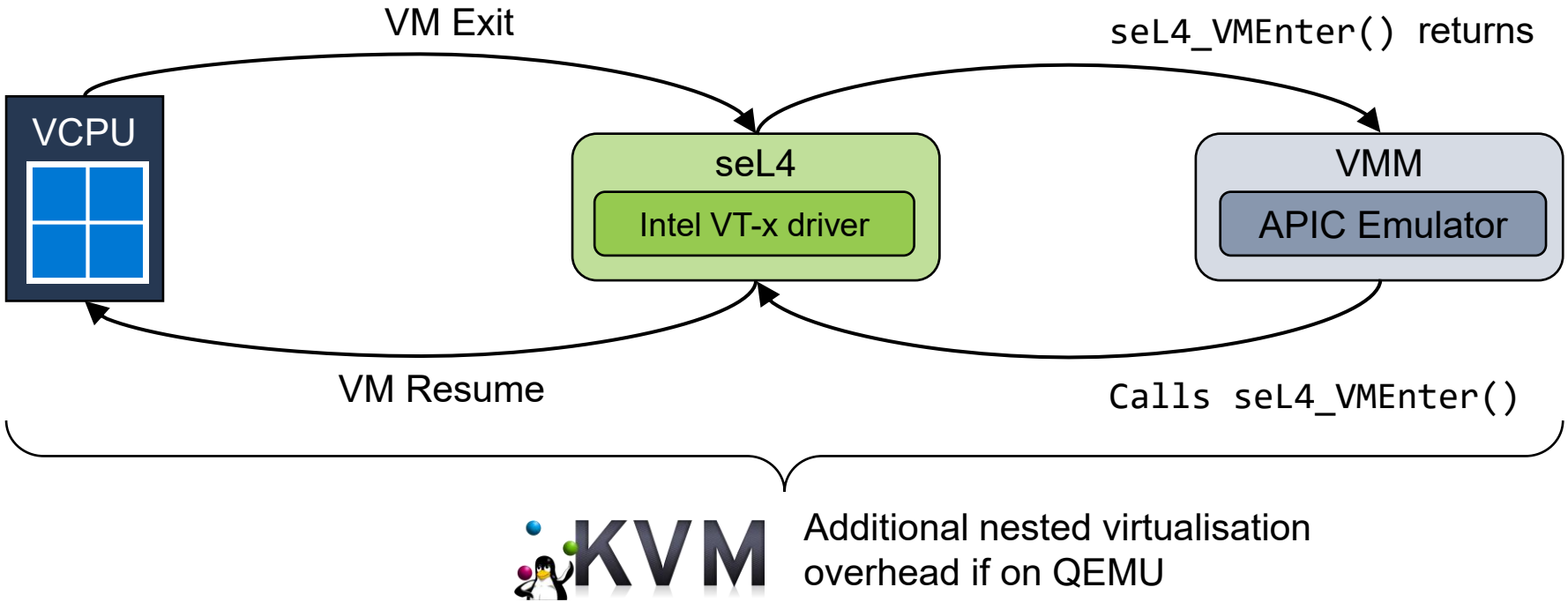
Intel APICv

- Linux used `cli` and `sti` which doesn't cause VM exits.



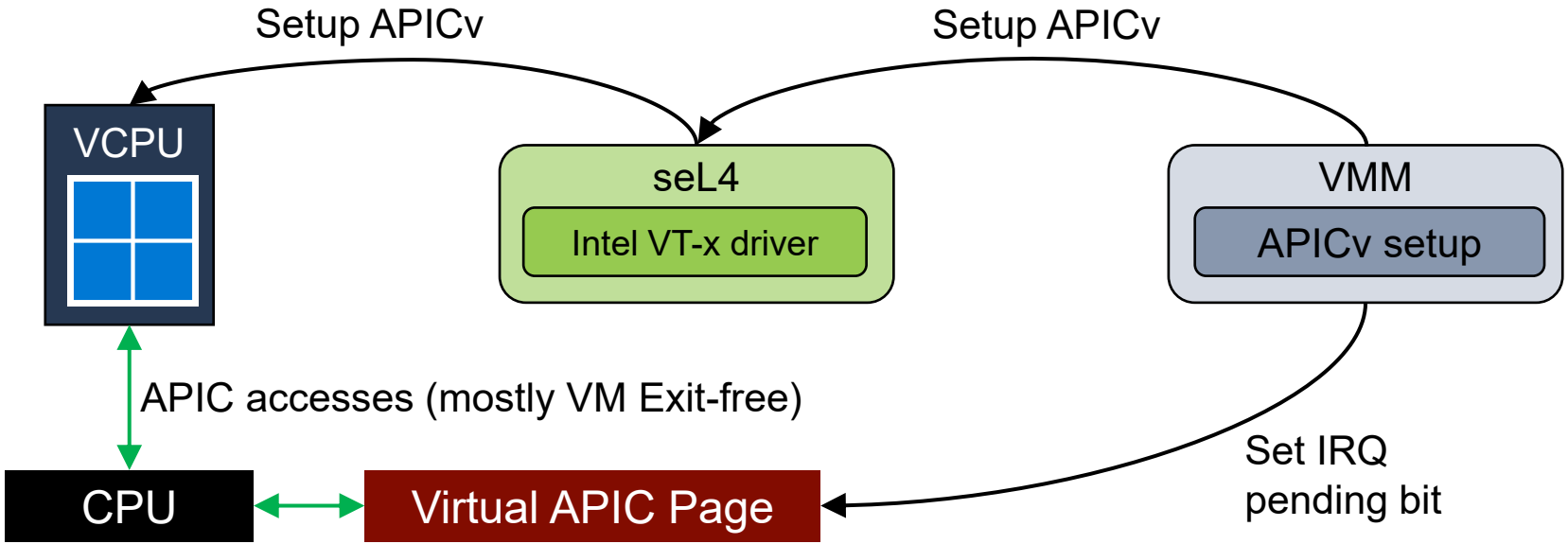
Intel APICv

- TPR access destroys guest performance:



Intel APICv

- Solution: use hardware acceleration.
- Hardware maintains virtual APIC (IRQ controller) state in a 4K page.
- Hardware performs most of APIC virtualisation rather than software.



Intel APICv

- Windows boot time went from >10 minutes to 60 seconds, desktop finally usable.
- Kernel RFC in discussions.
- Prior Art in Bhyve (BSD hypervisor): reviews.freebsd.org/D22942

Debugging

- Firmware made Windows bootable.
- APICv made it usable.
- **But how to debug a closed source guest that tells you nothing?**

Debugging - Linux

- Quite straightforward

  Kernel crash prints a stacktrace

```
[    0.070729] Oops: general protection fault, kernel NULL pointer dereference 0x0: 0000 [#1] SMP PTI
[    0.071297] CPU: 0 UID: 0 PID: 0 Comm: swapper/0 Not tainted 6.19.0 #1 PREEMPT(voluntary)
[    0.071680] RIP: 0010:fpu__init_cpu_xstate+0x75/0x100
[    0.071680] Code: 7f 01 b9 c4 01 00 00 48 89 c2 48 c1 ea 20 0f 30 66 90 65 48 89 05 23 14 45 02 48
[    0.071680] RSP: 0000:ffffffff97c03e70 EFLAGS: 00000246
[    0.071680] RAX: 0000000000000003 RBX: 0000000000000282 RCX: 0000000000000000
[    0.071680] RDX: 0000000000000000 RSI: 0000000000000000 RDI: 0000000003506f0
[    0.071680] RBP: 0000000000000200 R08: 0000000000000005 R09: ffffdfffffffffff
[    0.071680] R10: 0000200000000000 R11: 0000000000000014 R12: 0000000000000003
[    0.071680] R13: 0000000000000001 R14: 0000000000000000 R15: 0000000003e18000
[    0.071680] FS:  0000000000000000(0000) GS:ffffa11437111000(0000) knlGS:0000000000000000
[    0.071680] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[    0.071680] CR2: ffffa113c9800000 CR3: 0000000008c30000 CR4: 0000000003506f0
[    0.071680] Call Trace:
[    0.071680]  <TASK>
[    0.071680]  fpu__init_system_xstate+0x1e3/0xa40
[    0.071680]  fpu__init_system+0x9d/0xc0
[    0.071680]  arch_cpu_finalize_init+0xe9/0x170
[    0.071680]  start_kernel+0x6fc/0x780
[    0.071680]  x86_64_start_reservations+0x24/0x30
[    0.071680]  x86_64_start_kernel+0xc8/0xd0
[    0.071680]  common_startup_64+0x13e/0x148
[    0.071680]  </TASK>
[    0.071680] Modules linked in:
[    0.071680] ---[ end trace 0000000000000000 ]---
```

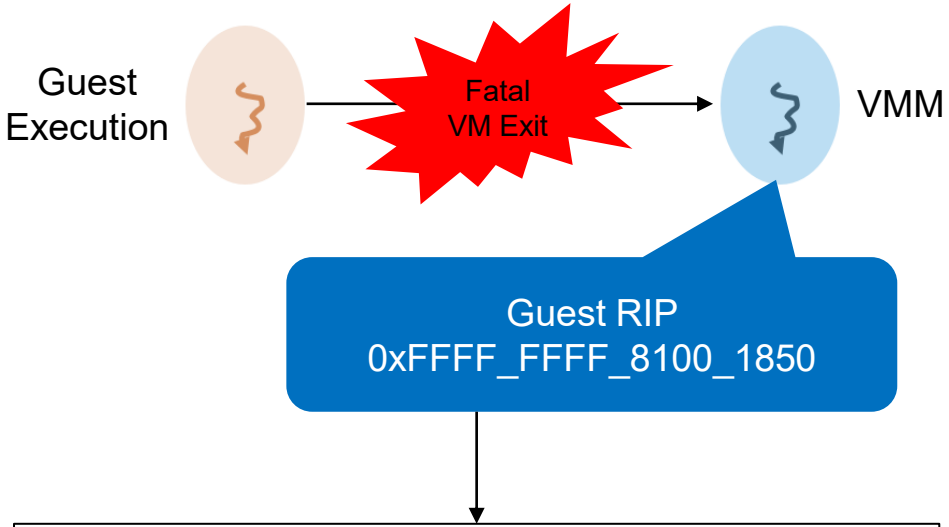
Debugging - Linux

- Quite straightforward

Kernel crash prints a stacktrace

```
[    0.070729] Oops: general protection fault, kernel NULL pointer dereference 0x0: 0000 [#1] SMP PTI
[    0.071297] CPU: 0 UID: 0 PID: 0 Comm: swapper/0 Not tainted 6.19.0 #1 PREEMPT(voluntary)
[    0.071680] RIP: 0010:fpu__init_cpu_xstate+0x75/0x100
[    0.071680] Code: 7f 01 b9 c4 01 00 00 48 89 c2 48 c1 ea 20 0f 30 66 90 65 48 89 05 23 14 45 02 48
[    0.071680] RSP: 0000:ffffffff97c03e70 EFLAGS: 00000246
[    0.071680] RAX: 0000000000000003 RBX: 0000000000000282 RCX: 0000000000000000
[    0.071680] RDX: 0000000000000000 RSI: 0000000000000000 RDI: 0000000003506f0
[    0.071680] RBP: 0000000000000200 R08: 0000000000000005 R09: ffffdfffffffffff
[    0.071680] R10: 0000200000000000 R11: 0000000000000014 R12: 0000000000000003
[    0.071680] R13: 0000000000000001 R14: 0000000000000000 R15: 0000000003e18000
[    0.071680] FS:  0000000000000000(0000) GS:ffffa11437111000(0000) knlGS:0000000000000000
[    0.071680] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[    0.071680] CR2: ffffa113c9800000 CR3: 000000008c30000 CR4: 0000000003506f0
[    0.071680] Call Trace:
[    0.071680]  <TASK>
[    0.071680]  fpu__init_system_xstate+0x1e3/0xa40
[    0.071680]  fpu__init_system+0x9d/0xc0
[    0.071680]  arch_cpu_finalize_init+0xe9/0x170
[    0.071680]  start_kernel+0x6fc/0x780
[    0.071680]  x86_64_start_reservations+0x24/0x30
[    0.071680]  x86_64_start_kernel+0xc8/0xd0
[    0.071680]  common_startup_64+0x13e/0x148
[    0.071680]  </TASK>
[    0.071680] Modules linked in:
[    0.071680] ---[ end trace 0000000000000000 ]---
```

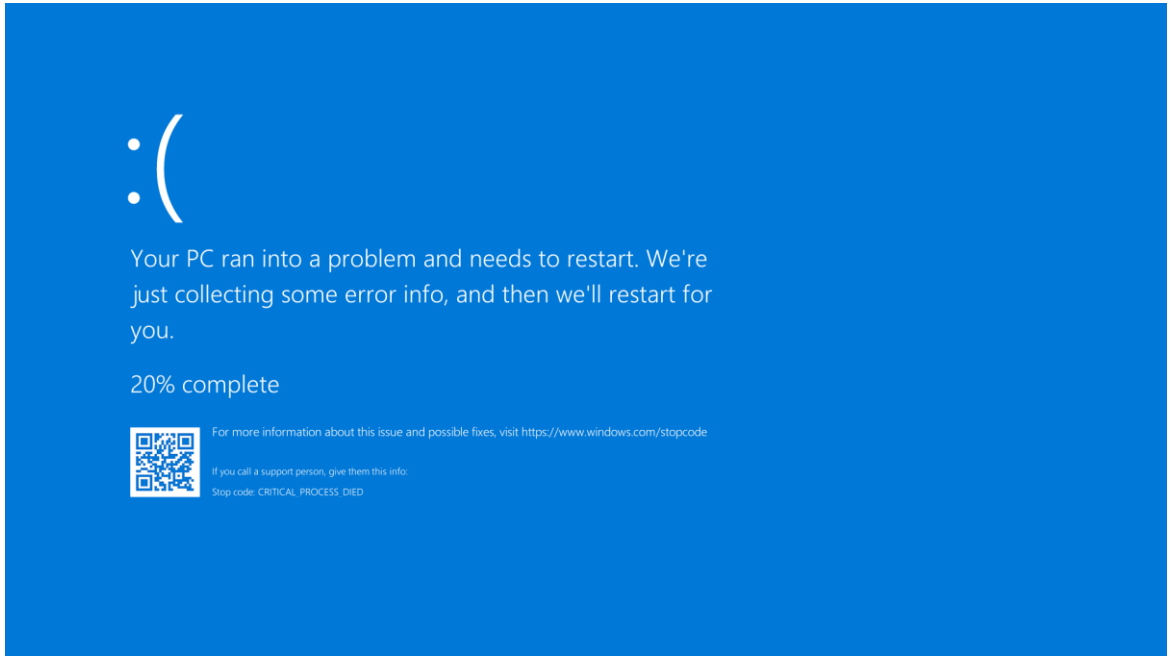
In other cases, turn off KASLR then:




```
$ addr2line -e vmlinux -f -i -C $RIP
…
…
# or other tools
```

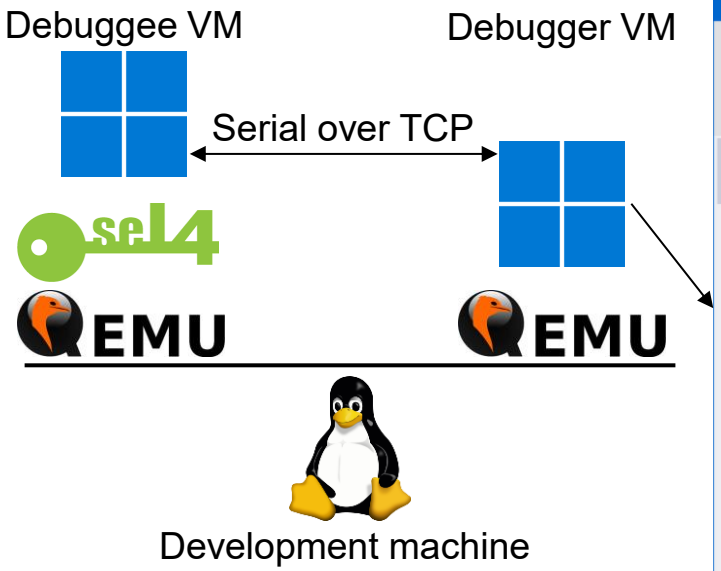
Debugging - Windows

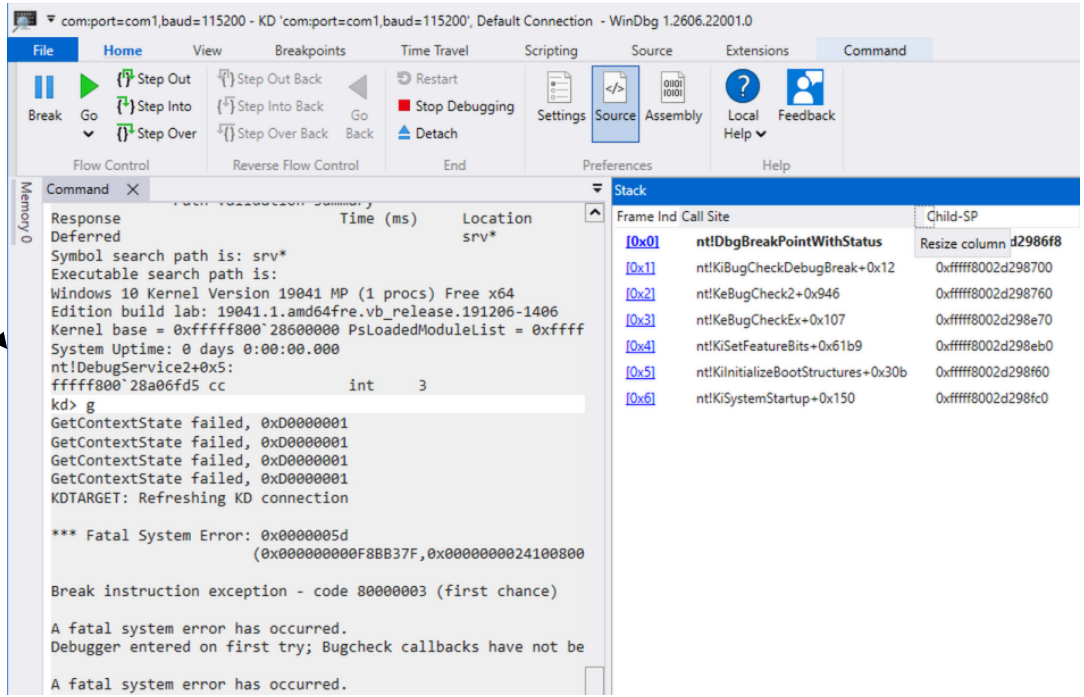
- Less forgiving. A problem does not cause a blue screen if the kernel has not booted far enough. Regardless, Windows doesn't give us enough information.



Debugging - Windows

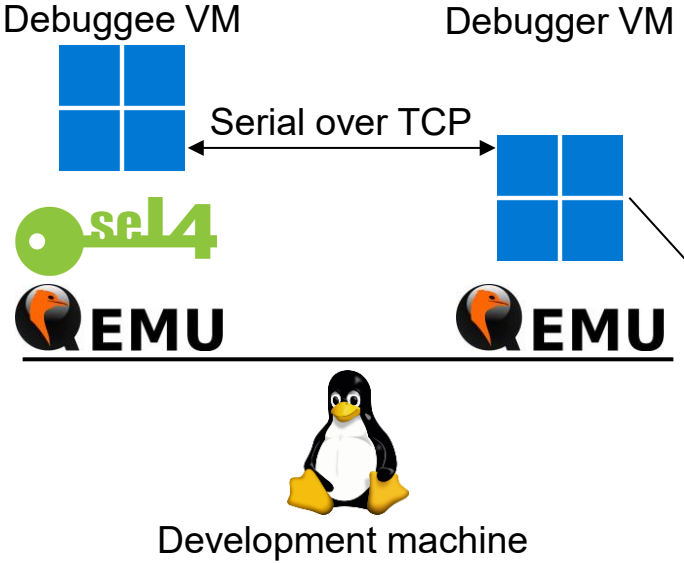
- Must use a kernel debugger and no C sources to look at.





Debugging - Windows

- Must use a kernel debugger and no C sources to look at. We do get some useful information:



Stack

Frame Ind	Call Site	Child-SP
[0x0]	nt!DbgBreakPointWithStatus	Resize column d2986f8
[0x1]	nt!KiBugCheckDebugBreak+0x12	0xfffff8002d298700
[0x2]	nt!KeBugCheck2+0x946	0xfffff8002d298760
[0x3]	nt!KeBugCheckEx+0x107	0xfffff8002d298e70
[0x4]	nt!KiSetFeatureBits+0x61b9	0xfffff8002d298eb0
[0x5]	nt!KiInitializeBootStructures+0x30b	0xfffff8002d298f60
[0x6]	nt!KiSystemStartup+0x150	0xfffff8002d298fc0

Debugging - Windows

- Problematic Windows kernel function identified, let's look at it under Ghidra!

CodeBrowser: w10_kernel:/ntoskrnl.exe

Decompile: KiSetFeatureBits - (ntoskrnl.exe)

```
195  if ((((((cpuid_01h_00h_edx & 0x789f3fd) != 0x789f3fd) || ((cpuid_80000001h_edx >> 0xb & 1) == 0))
196        || ((cpuid_80000001h_edx >> 0x14 & 1) == 0)) ||
197       (((cpuid_01h_00h_ecx & 0x2000) == 0 || ((cpuid_80000001h_ecx & 1) == 0)))) ||
198      (KiOpPrefetchPatchSkip != 0)) {
199    if (*(int *)(param_1 + 0x24) == 0) {
200      KdInitSystem(0,KeLoaderBlock);
201    }
202    uStack_88 = (ulonglong)KiOpPrefetchPatchSkip;
203                    /* WARNING: Subroutine does not return */
204    KeBugCheckEx(0x5d,cpuid_01h_00h_edx,cpuid_80000001h_edx,cpuid_80000001h_ecx);
205  }
```

Debugging - Windows

- Uh-oh we are missing CPU features. Fix, rinse and repeat.

CodeBrowser: w10_kernel:/ntoskrnl.exe

Decompile: KiSetFeatureBits - (ntoskrnl.exe)

```
195  if ((((cpuid_01h_00h_edx & 0x789f3fd) != 0x789f3fd) || ((cpuid_80000001h_edx >> 0xb & 1) == 0))
196        || ((cpuid_80000001h_edx >> 0x14 & 1) == 0)) ||
197       (((cpuid_01h_00h_ecx & 0x2000) == 0 || ((cpuid_80000001h_ecx & 1) == 0)))) ||
198      (KiOpPrefetchPatchSkip != 0)) {
199    if (*(int *)(param_1 + 0x24) == 0) {
200      KdInitSystem(0,KeLoaderBlock);
201    }
202    uStack_88 = (ulonglong)KiOpPrefetchPatchSkip;
203                    /* WARNING: Subroutine does not return */
204    KeBugCheckEx(0x5d,cpuid_01h_00h_edx,cpuid_80000001h_edx,cpuid_80000001h_ecx);
205  }
```

Live Demo?

Live Demo?

I lied.

Live Demo?

I lied.

You've been watching the demo the entire time.

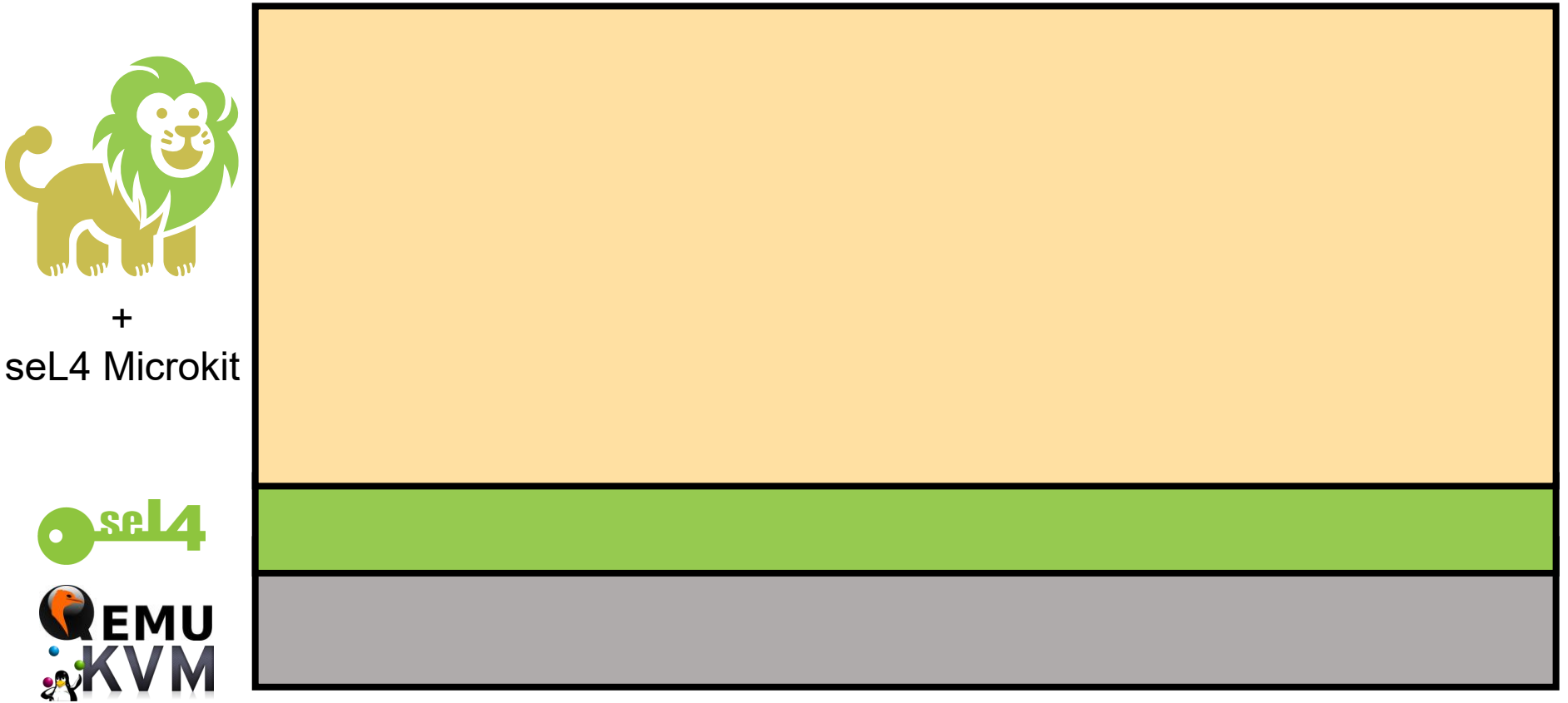
Live Demo?

I lied.

You've been watching the demo the entire time.

I wrote and ran my talk on libvmm.

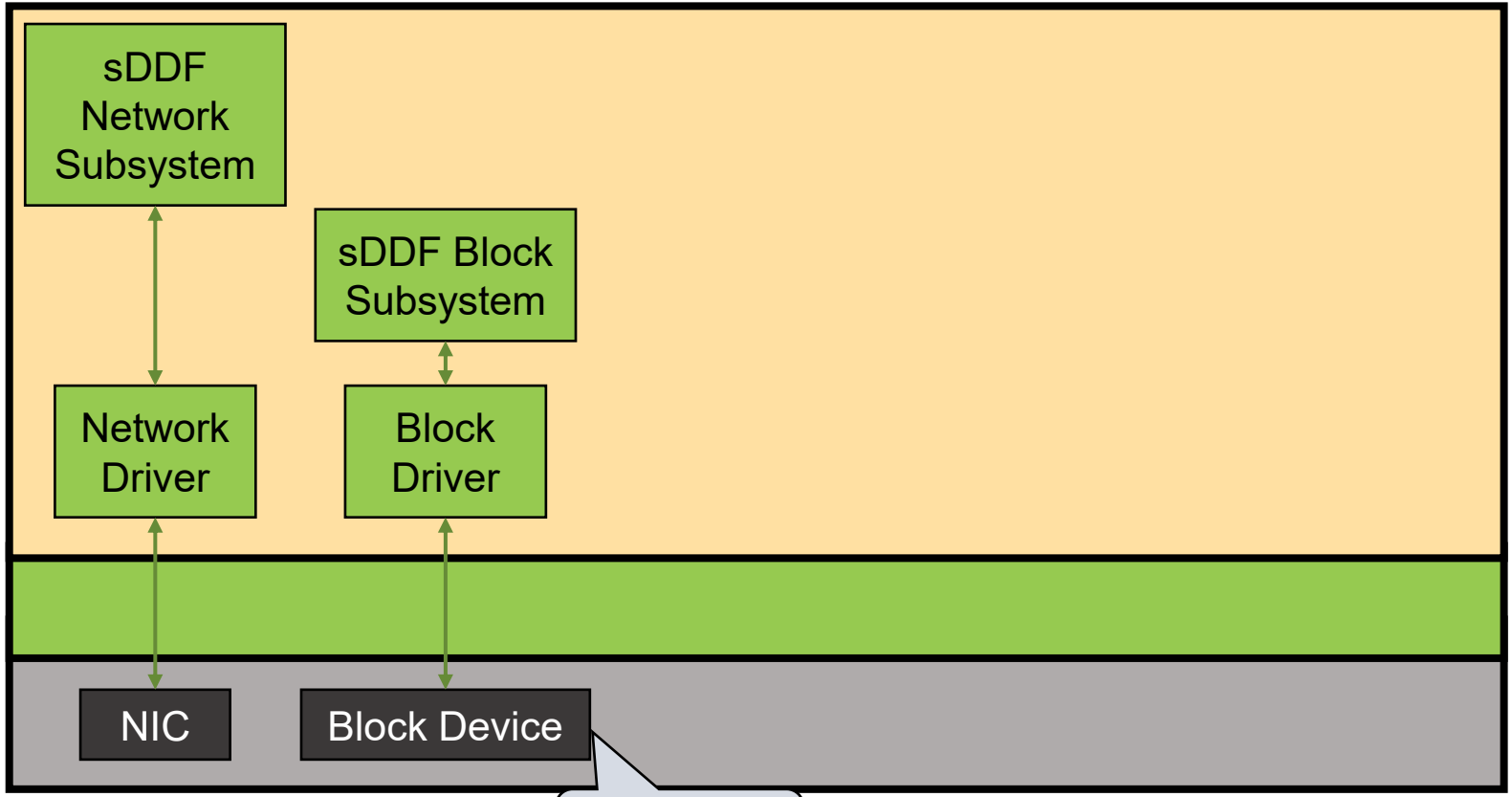
Demo architecture



+

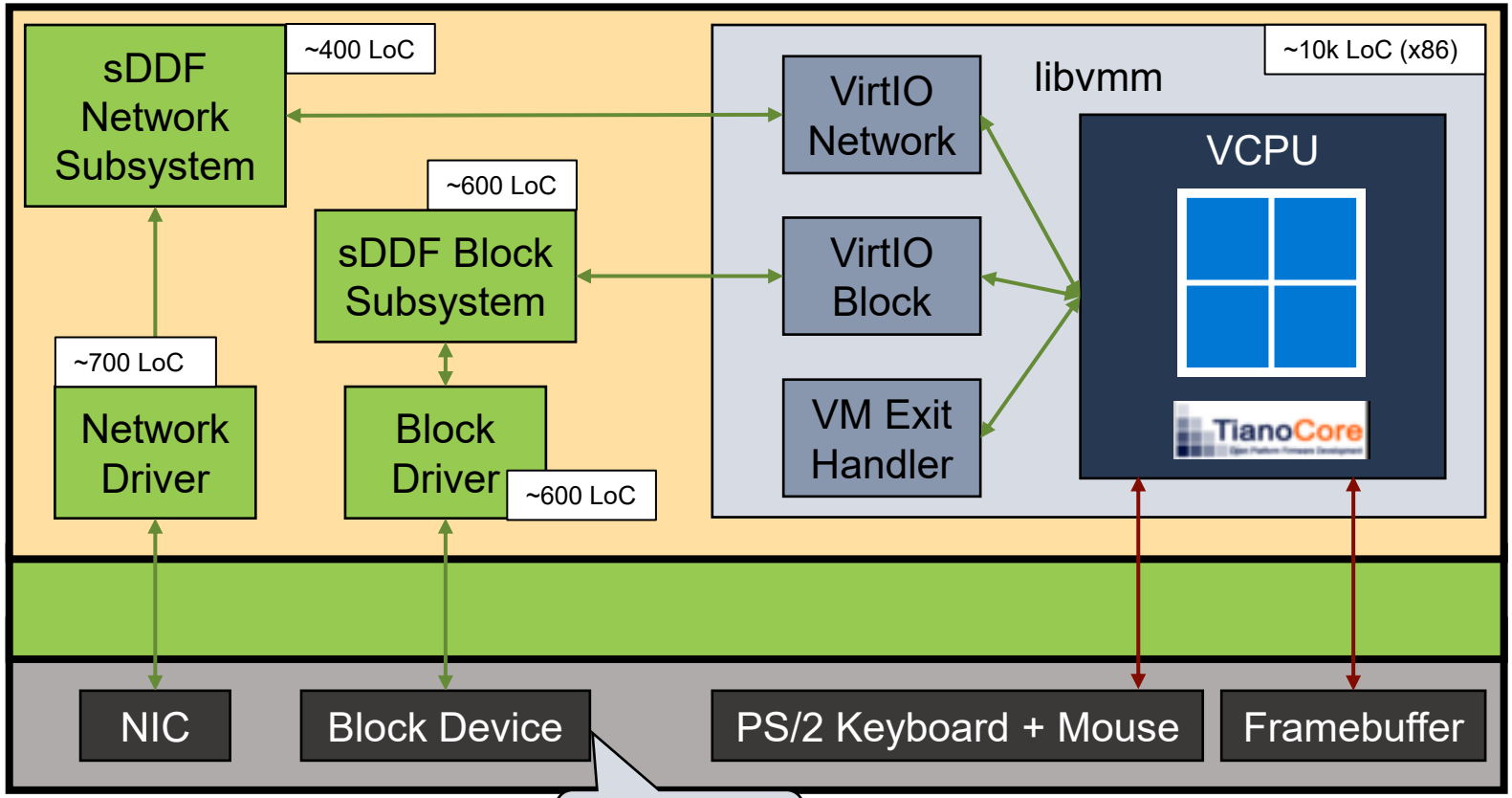
seL4 Microkit

Demo architecture



+
seL4 Microkit

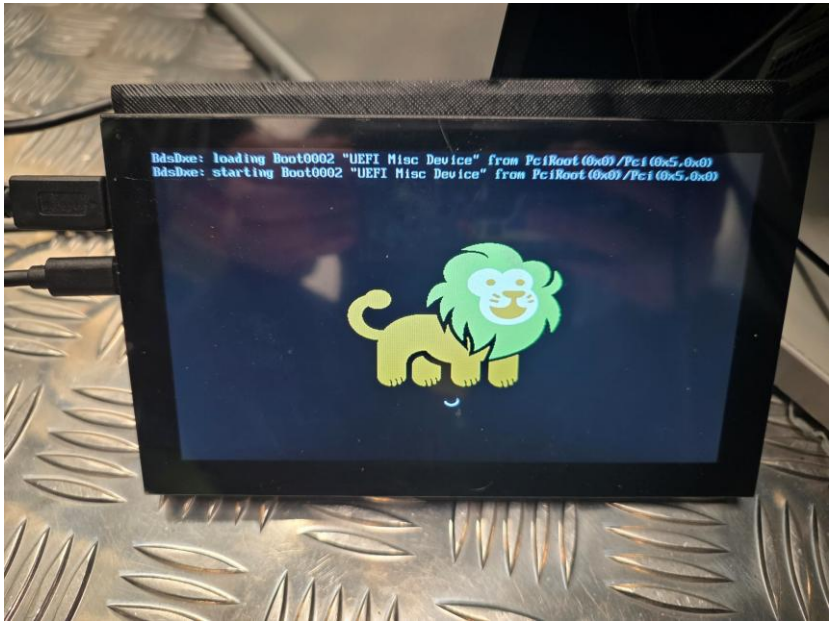
Demo architecture



+

seL4 Microkit

Works on hardware too



Intel Xeon W-1250, Windows loaded from an NVMe SSD with native sDDF driver.

Liked what you saw?

- You can take it home with you:

```
$ git clone https://github.com/au-ts/libvmm.git
$ cd libvmm
$ git checkout windows_mark_ii
$ git submodule update --init
$ cd examples/windows
$ less README.md
```

- Feel free to contribute!

Thank you for your time, questions?

Appendix A. libvmm Features Matrix

	aarch64	x86-64
Boot raw Linux kernel and initrd images	✅	✅
In-guest storage and networking with virtIO	✅	✅
Multi-core guests (> 1 VCPUs)	✅	
Boot UEFI firmware		✅
Boot Windows 10, 11 and Linux distributions (Ubuntu, etc.)		✅