MicroVMs From The Bottom Up

A reader-grade tour of KVM, virtio, and the Firecracker microVM, built from the CPU's virtualization extensions up.

Contents

PART I — ORIENTATION

Chapter 1: The Virtualization Stack Map

Running `lsmod | grep kvm` on a modern Linux host shows three modules: `kvm`, and either `kvm_intel` or `kvm_amd`. Those three lines are a complete hypervisor. Knowing what each one does, and what each one deliberately does not do, is the prerequisite for everything that follows.

People describe KVM as a type-2 hypervisor because a host OS is present; others call it type-1 because it runs in ring 0. Firecracker is described as "lightweight QEMU," which is technically wrong in almost every dimension that matters. QEMU is treated as inevitable, when it is one of four VMMs that share the same bottom two layers. The stack has a precise shape, and the vocabulary gets much less slippery once you can see the shape.

The Hardware Foundation

The first problem with running a guest operating system on a host is rings. An operating system assumes it owns ring 0. It writes control registers, flushes TLBs, modifies the GDT. If two operating systems try to do those things on the same hardware simultaneously, they collide. Classic x86 made this worse: many instructions behave differently in ring 0 than in ring 3 without trapping — they silently fail or return host state to the guest — which means a naive hypervisor cannot intercept them. The 1974 Popek-Goldberg paper in *Communications of the ACM* proved that virtualization requires all **sensitive** instructions (those whose behavior depends on privilege level) to be a subset of **privileged** instructions (those that trap in user mode). Classic x86 violated that condition for eighteen instructions.[^robin-irvine]

[^robin-irvine]: John D. Robin and Cynthia E. Irvine, "Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor," USENIX Security Symposium, 2000. The paper identifies eighteen sensitive, unprivileged IA-32 instructions.

Intel VT-x and AMD-V broke through that impasse not by fixing the ring semantics but by adding an orthogonal dimension above them. Intel's answer is two execution contexts: **VMX root mode**, where the host OS and KVM run, and **VMX non-root mode**, where the guest OS runs. Rings 0 through 3 continue to exist inside each mode. A guest kernel executes at VMX non-root ring 0; its applications at VMX non-root ring 3. The silicon, not software, enforces the boundary. When guest code executes something that requires host attention, the hardware performs a **VM exit** and transfers control to KVM's exit handler. The guest resumes when KVM issues `VMRESUME`. The Intel SDM (Vol. 3, Chapter 23) is the primary specification; AMD's equivalent mode is SVM (Secure Virtual Machine), enabled by the `EFER.SVME` bit.

Per-vCPU state on Intel lives in the **VMCS** (Virtual Machine Control Structure), a 4 KiB in-memory structure accessible only through the `VMREAD` and `VMWRITE` instructions, which are executable only in VMX root mode. The VMCS has six logical regions: a guest-state area saved on every VM exit and restored on every VM entry; a host-state area restored on exit; VM-execution control fields that specify which guest events trigger exits; VM-exit control fields; VM-entry control fields; and VM-exit information fields that

record what happened. The field `VM_EXIT_REASON` lives at VMCS encoding `0x00004402`. The EPT page-table pointer is at `EPT_POINTER = 0x0000201a`. These are not abstractions — they are the numeric encodings in `arch/x86/include/asm/vmx.h`, read by KVM with `VMREAD` after every exit.

AMD's counterpart is the **VMCB** (Virtual Machine Control Block), a packed struct defined in `arch/x86/include/asm/svm.h`:

```
struct __attribute__((__packed__)) vmcb {
    struct vmcb_control_area control;
    struct vmcb_save_area    save;
};
```

The entry instruction is `VMRUN`, which takes the guest VMCB's physical address in `RAX`. On `VMEXIT`, only `RIP`, `RSP`, and `RAX` are automatically restored to host values; all other general-purpose registers still hold guest values and must be explicitly saved by the hypervisor, typically with `VMLOAD`. The `control` area carries `exit_code`, `exit_info_1`, `exit_info_2`, and `next_rip`; the `save` area preserves the full segment and register state the guest had on entry.

Memory presents a parallel problem. A guest manages page tables that map its virtual addresses to what it believes are physical addresses. Those "physical" addresses are not host physical addresses — the guest does not own the machine's memory. Intel EPT (Extended Page Tables) and AMD NPT (Nested Page Tables) solve this by adding a second hardware translation: guest-physical address → host-physical address, walked by the MMU automatically on every guest memory access. KVM calls this TDP, two-dimensional paging. Without EPT or NPT, KVM falls back to **shadow page tables**: a `struct kvm_mmu_page` containing 512 shadow PTEs with `role.level` (1 = 4 KiB, 2 = 2 MiB, 3 = 1 GiB) and a `role.direct` flag. Shadow tables require KVM to write-protect all guest page tables and synchronize shadow PTEs on every guest modification — expensive enough that the fallback is for ancient or constrained environments. The KVM MMU documentation at `docs.kernel.org/virt/kvm/x86/mmu.html` documents both paths; on any server CPU built after roughly 2008, EPT or NPT is the operative path.

The KVM Kernel Module

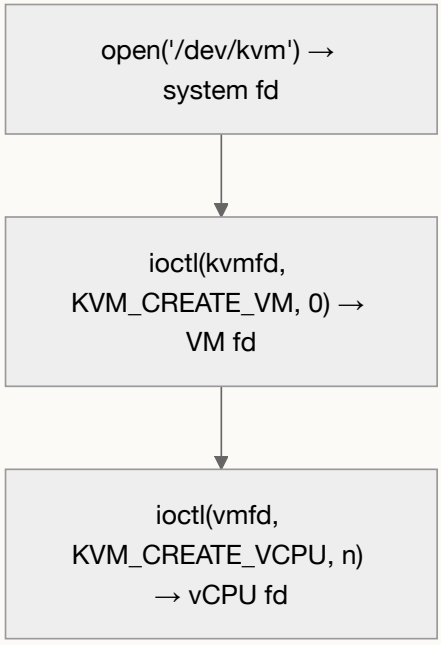
The hardware extensions provide the mechanism. KVM makes it programmable.

KVM ships as three kernel modules. `kvm.ko` is the common base: it creates `/dev/kvm`, owns the ioctl dispatch table, manages memory slots, runs the in-kernel irqchip (LAPIC, IOAPIC, PIT, HPET via `kvm_io_bus`), and sets up EPT or NPT. `kvm-intel.ko` implements the VMX code path and requires the `vmx` CPU feature bit. `kvm-amd.ko` implements the SVM path and requires `svm`. Architecture-specific operations flow through the `kvm_x86_ops` function-pointer struct (`struct kvm_x86_ops`); the common module calls whichever vendor implementation is loaded. Minimum requirements: Linux 4.14 or later with `CONFIG_KVM=m` and `CONFIG_KVM_INTEL=m` or `CONFIG_KVM_AMD=m`.

`/dev/kvm` is a character device with mode `0660`, owned `root:kvm`. Its major number is dynamically assigned and visible in `/proc/devices`. Any process in the `kvm` group can open it; everything else is mediated through the ioctl API above it.

The Three-Level File Descriptor API

The KVM API is organized as three nested file descriptors, each narrowing scope from the kernel-wide to the per-CPU:



The system fd talks to KVM as a whole. `KVM_GET_API_VERSION` (`_IO(0xAE, 0x00)`) returns the constant `12`, a value frozen at Linux 2.6.22; any value other than 12 indicates a broken kernel and must be rejected. `KVM_CREATE_VM` (`_IO(0xAE, 0x01)`) returns a VM fd; the parameter is the machine type, `0` for the default x86 type.

The VM fd talks to one virtual machine. `KVM_SET_USER_MEMORY_REGION` (`_IOW(0xAE, 0x46, ...)`) maps a range of host virtual memory into the guest's physical address space:

```
struct kvm_userspace_memory_region {
    __u32 slot;            /* region index, bits 0–15 */
    __u32 flags;           /* KVM_MEM_LOG_DIRTY_PAGES | KVM_MEM_READONLY */
    __u64 guest_phys_addr;
    __u64 memory_size;     /* bytes; set to 0 to delete the slot */
    __u64 userspace_addr;  /* host virtual address of backing memory */
};
```

The `userspace_addr` field is the insight about guest "physical" memory: it is a host virtual address, typically obtained with `mmap`. EPT then walks from that host virtual address to the true host-physical page. Slots must not overlap in guest physical space. `KVM_CREATE_VCPU` (`_IO(0xAE, 0x41)`) takes a vCPU ID (which becomes the APIC ID on x86) and returns a vCPU fd.

The vCPU fd talks to one virtual CPU. `KVM_SET_REGS` (`_IOW(0xAE, 0x82, ...)`) initializes general-purpose registers; `KVM_SET_SREGS` (`_IOW(0xAE, 0x84, ...)`) sets segment registers, control registers, and `EFER`. `KVM_RUN` (`_IO(0xAE, 0x80)`) enters the guest and blocks the calling thread until a VM exit requires userspace attention.

kvm_run and Exit Reasons

Before the first `KVM_RUN`, the VMM maps the `struct kvm_run` shared region:

```
mmap(NULL, mmap_size, PROT_READ|PROT_WRITE, MAP_SHARED, vcpufd, 0)
```

where `mmap_size` comes from `KVM_GET_VCPU_MMAP_SIZE` on the system fd. When `KVM_RUN` returns, `kvm_run.exit_reason` tells the VMM why. The exit codes are defined in `include/uapi/linux/kvm.h` and are part of the stable ABI:

Code	Name	Meaning
0	`KVM_EXIT_UNKNOWN`	Unhandled exit
2	`KVM_EXIT_IO`	Guest I/O port access; `io.direction`, `io.port`, `io.size`, `io.count`, `io.data_offset` populated
5	`KVM_EXIT_HLT`	Guest executed `HLT` with no pending work
6	`KVM_EXIT_MMIO`	Guest MMIO access; `mmio.phys_addr`, `mmio.data`, `mmio.len`, `mmio.is_write` populated
8	`KVM_EXIT_SHUTDOWN`	Guest triple-faulted or requested shutdown
17	`KVM_EXIT_INTERNAL_ERROR`	KVM internal error
24	`KVM_EXIT_SYSTEM_EVENT`	Reboot or poweroff

Not every VM exit reaches userspace. KVM's exit handler in `arch/x86/kvm/vmx/vmx.c` dispatches on `EXIT_REASON` through a table of per-reason handlers. `EXIT_REASON_CPUID = 10` is handled entirely in kernel; the handler synthesizes the correct leaf values and returns `1`, causing KVM to re-enter the guest immediately without returning to the VMM process. `EXIT_REASON_IO_INSTRUCTION = 30` goes in-kernel if the `kvm_io_bus` has a handler registered for that port; otherwise `KVM_RUN` returns with `KVM_EXIT_IO`. `EXIT_REASON_EPT_VIOLATION = 48` is in-kernel for page-table faults KVM can resolve itself; if the access targets a region the VMM must emulate, `KVM_RUN` returns with `KVM_EXIT_MMIO`. The handler returning `1` is the fast path; `0` or a negative value sends the exit to userspace.

What KVM Owns, What the VMM Owns

The split is enforced by KVM, not advisory:

KVM owns: CPU context save and restore through the VMCS or VMCB; hardware interrupt injection; EPT and NPT management; the VM-exit dispatch loop; and the in-kernel irqchip (LAPIC, IOAPIC, PIT, HPET).

The VMM owns: device models; guest memory layout and the `mmap` calls that back it; firmware or its deliberate absence; and policy for vCPU thread scheduling. One threading constraint shapes every VMM: `KVM_RUN` and all other vCPU ioctls must be issued from the same thread that called `KVM_CREATE_VCPU`. VM-level ioctls must come from the same process that created the VM. KVM enforces this; it is not advisory.

Type 1, Type 2, and Why the Labels Break Down

The type-1/type-2 taxonomy that opens every virtualization talk traces to Robert P. Goldberg's 1972 Harvard PhD thesis, not to Popek and Goldberg's 1974 *Communications of the ACM* paper. The 1974 paper defines three VMM properties — equivalence/fidelity, resource control/safety, and efficiency/performance — and proves the formal conditions for VMM construction. It never uses the terms "Type 1" or "Type 2." Those labels came from the thesis: a **Type I** VMM runs on bare hardware; a **Type II** VMM runs on top of a host OS, an "extended machine."

KVM satisfies both definitions simultaneously, which is the problem. The KVM module runs at VMX root ring 0, the same privilege level as the host Linux kernel — both are in VMX root mode at ring 0. That is Goldberg's Type I condition: supervisor privilege on hardware. But a host OS runs alongside and beneath the VMM; the same machine runs host applications concurrently with guest VMs. That is the structural situation the Type II label was invented to describe.

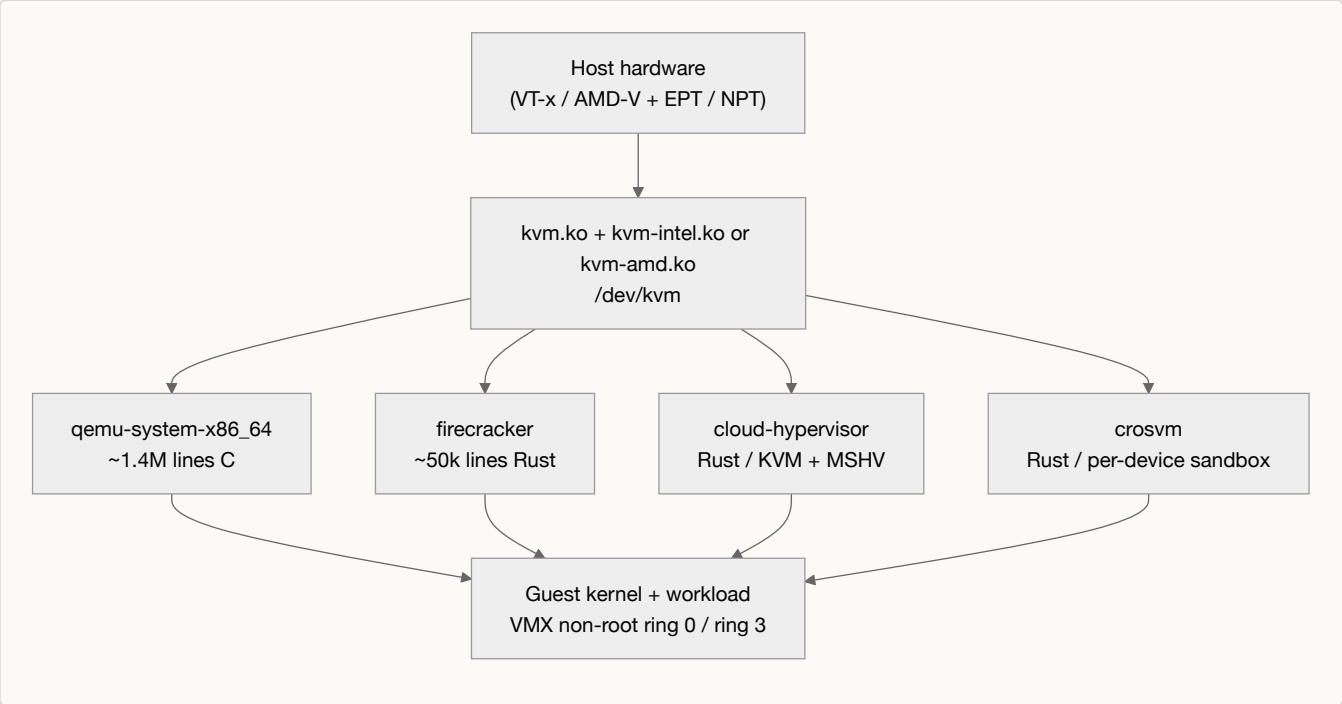
KVM maintainer Paolo Bonzini addressed the question directly in a thread on LKML: "I would just ignore it. To some extent, the modern usage of the type-1 and type-2 terms is really more about VMware and Xen trying to bash KVM, than anything else." Kernel developer Christoph Hellwig, on the same thread, called the arbitrary classification something that "doesn't make any sense at all."

The structural reason both are right: VMX root mode is orthogonal to the ring hierarchy. There is no longer a meaningful "above the OS" or "below the OS" position. The host kernel and the KVM module coexist at VMX root ring 0; guests run at VMX non-root ring 0. The 1972 taxonomy predates this hardware entirely. Xen with Dom0 is structurally identical to KVM with QEMU — hypervisor core at bare-metal privilege, a privileged user-mode component for device emulation — and neither label fits either system cleanly.

The only accurate description is **hybrid**: Type I in hardware privilege, Type II in host coexistence. This book treats it that way and does not return to the question.

The VMM Landscape

With the hardware and KVM layers established, the VMMs are easier to place. All of them are userspace processes that open `/dev/kvm`, call the three-level ioctl API, back guest memory with `mmap`, and loop on `KVM_RUN`. What they differ in is what they model after that — which devices, which transports, which security boundaries, which guest configurations they serve.



QEMU

Fabrice Bellard released the first QEMU preview in 2003. The codebase now stands at approximately 1.4 million lines of C, supports more than 30 guest ISAs, and runs under two acceleration modes. **TCG** (Tiny Code Generator) is a software JIT that cross-compiles guest instructions to host instructions at runtime; it is the default and requires no hardware support. Hardware acceleration — KVM on Linux, Hypervisor.Framework on macOS, WHPX on Windows, Xen — is selected explicitly. When you run `qemu-system-x86_64 -accel kvm`, the guest vCPUs execute directly on hardware through `/dev/kvm`; TCG does none of the work.

QEMU's breadth is its defining characteristic. It emulates PCI buses, USB controllers, AHCI storage, GPU variants, ISA buses, legacy interrupt controllers, BIOS and UEFI firmware, and hundreds of individual device models. That breadth is also the attack surface. QEMU does ship a minimal machine type called `microvm` (selected with `-machine microvm`), directly inspired by Firecracker: it drops PCI and ACPI, exposes up to eight virtio-MMIO devices, one optional ISA serial port, a LAPIC, an IOAPIC, kvmclock (when using KVM), and fw_cfg. It cannot hotplug devices or live-migrate across QEMU versions.

Intel's earlier attempt at the same goal, the NEMU project (`github.com/intel/nemu`), was archived on April 14, 2021. Its archived README reads: "Cloud Hypervisor is the successor."

Firecracker

AWS open-sourced Firecracker on November 27, 2018, at version 0.11.0. It is written in approximately 50,000 lines of Rust — a 96% reduction from QEMU's codebase — and is licensed Apache 2.0. It supports x86_64 and aarch64, and runs exclusively on KVM with no TCG fallback.

Firecracker is a direct descendant of crosvm. The crosvm documentation states this explicitly: "Soon after crosvm's code was public, Amazon used it as the basis for their own VMM named Firecracker." Cloud Hypervisor's README likewise acknowledges that "a large part of Cloud Hypervisor code is based on either the Firecracker or crosvm implementations."

The process structure is fixed: one process per microVM. Inside that process, three thread types exist: an API thread running an in-process HTTP/REST server over a Unix domain socket, a VMM thread handling device emulation and I/O rate limiting, and one `KVM_RUN`-looping vCPU thread per guest vCPU. The device set is intentionally small. Firecracker emulates `virtio-net`, `virtio-block`, `virtio-vsock`, `virtio-balloon` (added in v1.4.0), a serial console (16550A UART), and an i8042 keyboard controller stub. All virtio devices use the **virtio-over-MMIO** transport defined in OASIS virtio spec §4.2, not virtio-over-PCI. Interrupt controllers — two 8259 PICs and an IOAPIC — are required for x86 interrupt delivery and are emulated via KVM. On aarch64, a PL031 RTC is also present. No USB, no display, no audio, no GPU.

That fixed structure enables Firecracker's `SPECIFICATION.md` to make enforceable quantitative claims — bounds that run as integration tests on every pull request and main-branch merge. VMM start to API socket availability: at most 8 CPU milliseconds. Guest `/sbin/init` reachable from `InstanceStart`: at most 125 ms. VMM memory overhead for a 1-vCPU, 128 MiB guest: at most 5 MiB. Guest CPU compute performance: above 95% of bare-metal equivalent. These are not marketing targets; they are pass/fail CI gates.

Cloud Hypervisor

Intel announced Cloud Hypervisor in May 2019, grown from NEMU. It targets cloud VM workloads — not ultra-minimal serverless — and its feature set reflects that: CPU and memory hotplug, VFIO device passthrough for SR-IOV NICs and NVMe drives, `virtio-fs`, `virtio-pmem`, `vhost-user` for device backend offload, and Intel TDX confidential-VM support. It supports 64-bit Linux guests and Windows 10 and Windows Server 2019 guests.

Cloud Hypervisor runs on both Linux KVM and the Microsoft Hypervisor (MSHV), enabling deployment on Azure and Windows Hyper-V hosts. Primary architectures are x86-64 and AArch64 (production, requiring GICv3); riscv64 support is experimental.

crosvm

Google's crosvm was built for ChromeOS (the Crostini Linux container runtime) and Android (ARCVM). It is written primarily in Rust and runs primarily on KVM.

crosvm's defining architectural choice is a **process-per-device security model**. Each virtio device backend runs as a sandboxed child process, jailed with `minijail` — Google's library wrapping Linux namespaces and seccomp-BPF. A compromised device process is contained by a seccomp policy restricting its available syscalls; it cannot reach the rest of the VMM. This is structurally distinct from Firecracker's model, where seccomp-BPF filters are installed per-thread within a single process. crosvm's device set is broader: block, networking, memory ballooning, `virtio-fs`, vsock, `virtio-pmem`, USB, Wayland display output, experimental video, and `vhost-user`.

The rust-vmm Common Foundation

Firecracker, Cloud Hypervisor, and crosvm all independently implement the same pieces: KVM ioctl wrappers, guest memory management, virtio queue logic, legacy device models. The rust-vmm project, founded in December 2018 by engineers from Amazon, Google, Intel, and Red Hat — with Alibaba, CrowdStrike, and Linaro joining later — exists to factor those pieces into shared crates. All three Rust VMMs are named primary consumers.

The key crates, all published on crates.io: `kvm-ioctls` (safe Rust wrappers over the three-level KVM fd API); `kvm-bindings` (FFI bindings to the KVM UAPI headers); `vm-memory` (v0.17.1, October 2025, providing `GuestMemoryMmap` and `GuestAddress`, decoupling memory consumers from the allocation strategy); `linux-loader` (loads `vmlinux` ELF, `bzImage`, and PE images into guest memory; parses `boot_params` and `hvm_start_info`); `vm-superio` (emulates the UART 16550A, the i8042 PS/2 controller, and the ARM PL031 RTC); `seccompiler` (compiles seccomp-BPF filter policies and installs them per-thread via `apply_filter()`); `virtio-queue` (split-ring virtqueue implementation from the virtio spec). When you read Firecracker source and find the serial port coming from `vm-superio` and the KVM calls going through `kvm-ioctls`, you are looking at the shared layer.

Where Firecracker Sits and What It Leaves Out

Firecracker's design document states the purpose directly: "purpose-built for running serverless functions and containers safely and efficiently, and nothing more." Every omission reduces either boot latency, memory footprint, or attack surface — usually all three.

No BIOS, No Firmware

A conventional x86 boot starts in 16-bit real mode with firmware code from a ROM. The firmware probes hardware, initializes memory, runs the bootloader, and eventually hands off to the kernel. On a good day this takes hundreds of milliseconds and is necessary only to accommodate the diversity of hardware a general-purpose machine might encounter. A microVM has none of that diversity. Firecracker bypasses firmware entirely using the Linux x86 direct-boot protocol.

The protocol, documented at `docs.kernel.org/arch/x86/boot.html`, requires the VMM to write a populated `struct boot_params` — the "zero page" — into guest memory, set `%rsi` to point to it, and jump directly to the protected-mode kernel entry point. No 16-bit real-mode code executes. The

`setup_header` embedded in `boot_params` must carry the magic value `0x53726448` ("HdrS") at byte offset `0x202`, the protocol version at `0x206`, the bootloader type at `0x210`, load flags at `0x211`, the physical address of the kernel command line at `0x228`, and the linear memory required during init at `0x260`. Firecracker places the zero page at guest physical address `0x7000`; the `linux-loader` rust-vmm crate handles the mechanics of populating it.

Firecracker v1.12.0 (PR #5048) added support for the Xen PVH boot protocol. In PVH mode, the kernel must be compiled with `CONFIG_PVH=y` (available since Linux 5.0); the ELF binary carries a PVH entry point in a `PT_NOTE` segment, and the VMM passes an `hvm_start_info` structure with the memory map. The ACPI RSDP is placed at guest physical `0x000E_0000`. PVH is a second firmware-free boot path for kernels that prefer the cleaner memory-map handoff it provides.

Amazon's own description is blunt: "Firecracker doesn't implement traditional devices like a BIOS or PCI bus."

No PCI Bus

Firecracker has no PCI bus controller. All virtio devices use the virtio-over-MMIO transport (virtio 1.2 spec §4.2), not virtio-over-PCI. Each device occupies a fixed MMIO address range. The MMIO register map is defined by the spec: a `MagicValue` register at offset `0x000` containing `0x74726976` ("virt" in little-endian ASCII), a `Version` register at `0x004` holding `2` (the non-legacy virtio 1.x value), `DeviceID` at `0x008`, `VendorID` at `0x00C`, `DeviceFeatures` at `0x010`, and `QueueNotify` at `0x050`.

Unlike PCI, the MMIO transport has no generic discovery mechanism. Device locations must be communicated to the guest out-of-band. Historically Firecracker used kernel command-line slugs of the form `virtio_mmio.device=<size>@<addr>:<irq>`, which requires `CONFIG_VIRTIO_MMIO_CMDLINE_DEVICES` in the guest kernel. Since Firecracker v1.8.0, ACPI DSDT AML tables describe each virtio device, making the command-line approach unnecessary for ACPI-capable guests. A community initiative for PCIe passthrough exists (GitHub discussions #4845) but has no production release date; internal Firecracker team meetings on the topic were paused as of early 2025.

ACPI: Added Late, Not Original

ACPI was absent from Firecracker's original design. The first hardware topology mechanism was the legacy MultiProcessor Table (MPTable), sufficient for SMP discovery. Firecracker v1.8.0 (PR #4428) added basic ACPI tables for x86_64: a MADT describing vCPU and interrupt topology, and a DSDT with AML describing virtio and legacy devices. MPTable is now deprecated, with removal planned for v2.0 or later.

Even with ACPI present, there is no ACPI power management. Firecracker's FAQ states this directly: "Firecracker does not virtualize power management (e.g. there is no ACPI PM support)." Running `poweroff` inside a Firecracker guest shuts the guest OS down but leaves the `firecracker` process running, because the guest has no channel to signal power-off to the host.

The i8042 Stub

The i8042 keyboard controller appears in Firecracker's device list not to accept keyboard input but to field guest reboot requests. The FAQ specifies that reboot works only when the guest boots with `reboot=k`, which instructs Linux to signal reset through the i8042 reset line. It is on the path to removal once an ACPI S5 shutdown path is fully wired. Its presence illustrates the pattern: every legacy device in Firecracker is present because the guest kernel expects it at that address, not because the use case requires it.

The Guest Side

The guest runs in VMX non-root mode and cannot directly observe that it is virtualized — sensitive instructions either trap to KVM or, for most memory accesses, are handled transparently by EPT without a VM exit at all. Firecracker's guest requirements are: 64-bit Linux 4.14 or later, with `CONFIG_VIRTIO_MMIO=y` for device drivers. Even though no PCI bus is present, `CONFIG_PCI=y` and `CONFIG_PCI_MMCONFIG=y` may be required for ACPI initialization code paths; the kernel command line should include `pci=off` to suppress PCI enumeration. Firecracker's kernel policy document tracks the full set of required and recommended configuration options.

The next chapter works through the KVM ioctl API by building the smallest possible VMM that can boot a kernel — which is also the fastest way to see exactly where the hardware layer ends and the VMM layer begins.

Sources And Further Reading

- Popek & Goldberg, "Formal Requirements for Virtualizable Third-Generation Architectures," *Communications of the ACM* 17(7), July 1974. Proves the formal construction conditions; does not define Type 1/2. https://dl.acm.org/doi/10.1145/361011.361073
- Wikipedia: Popek and Goldberg virtualization requirements — confirms the 1974 paper does not use "Type 1" or "Type 2"; traces the labels to Goldberg's 1972 Harvard PhD thesis. https://en.wikipedia.org/wiki/Popek_and_Goldberg_virtualization_requirements
- Intel Software Developer's Manual, Vol. 3, Chapter 23: VMX non-root operation, VMCS structure, VM-exit behavior.
- KVM API documentation — definitive reference for the three-level fd model, ioctl encodings, `kvm_run`, and exit codes. https://docs.kernel.org/virt/kvm/api.html
- "Using the KVM API," LWN.net — C-code walkthrough of the three-level fd API and `struct kvm_run`. https://lwn.net/Articles/658511/
- `include/uapi/linux/kvm.h` — KVMIO ioctl encodings and the full `KVM_EXIT_*` enum. https://github.com/torvalds/linux/blob/master/include/uapi/linux/kvm.h
- `arch/x86/include/asm/vmx.h` — VMCS field encodings (`EPT_POINTER = 0x0000201a`, `VM_EXIT_REASON = 0x00004402`).

https://github.com/torvalds/linux/blob/master/arch/x86/include/asm/vmx.h

- `arch/x86/include/uapi/asm/vmx.h` — `EXIT_REASON_*` numeric constants used by KVM's exit handler. https://github.com/torvalds/linux/blob/master/arch/x86/include/uapi/asm/vmx.h
- `arch/x86/include/asm/svm.h` — VMCB struct layout (`vmcb_control_area`, `vmcb_save_area`). https://github.com/torvalds/linux/blob/master/arch/x86/include/asm/svm.h
- `arch/x86/include/uapi/asm/svm.h` — SVM exit codes (`SVM_EXIT_NPF = 0x400`). https://github.com/torvalds/linux/blob/master/arch/x86/include/uapi/asm/svm.h
- KVM shadow MMU vs. EPT/NPT TDP documentation. https://docs.kernel.org/virt/kvm/x86/mmu.html
- KVM nested VMX documentation — vmcs01/vmcs12/vmcs02 model; enabled by default since Linux v4.20. https://docs.kernel.org/virt/kvm/x86/nested-vmx.html
- LKML thread — Bonzini and Hellwig on the type-1/type-2 classification. https://lkml.kernel.org/kvm/cd9386f6-91d0-45a6-7a0c-30c8b58ada8d@chicoree.fr/T/
- Firecracker open-source announcement (November 27, 2018, v0.11.0). https://aws.amazon.com/blogs/opensource/firecracker-open-source-secure-fast-microvm-serverless/
- Amazon Science: how Firecracker virtual machines work — omissions (BIOS, PCI, USB, display, audio), ~50k lines Rust. https://www.amazon.science/blog/how-awss-firecracker-virtual-machines-work
- Firecracker design document — thread model (API thread, VMM thread, vCPU threads), device set, isolation layers, seccomp threat model. https://github.com/firecracker-microvm/firecracker/blob/main/docs/design.md
- Firecracker `SPECIFICATION.md` — quantitative SLIs enforced as CI integration tests. https://raw.githubusercontent.com/firecracker-microvm/firecracker/main/SPECIFICATION.md
- Firecracker FAQ — device list, ACPI PM absent, i8042 reboot, `reboot=k`. https://raw.githubusercontent.com/firecracker-microvm/firecracker/main/FAQ.md
- Firecracker CHANGELOG — v1.4.0 virtio-balloon, v1.8.0 ACPI (PR #4428), v1.12.0 PVH boot (PR #5048), MPTable deprecation. https://github.com/firecracker-microvm/firecracker/blob/main/CHANGELOG.md
- Firecracker jailer documentation — cgroup v1/v2, `CLONE_NEWPID`, `setns(CLONE_NEWNET)`, `pivot_root`. https://github.com/firecracker-microvm/firecracker/blob/main/docs/jailer.md
- Linux x86 boot protocol — zero page, `HdrS` magic at offset `0x202`, `cmd_line_ptr` at `0x228`. https://docs.kernel.org/arch/x86/boot.html
- LWN Firecracker article — direct boot, no BIOS, no firmware. https://lwn.net/Articles/775736/
- OASIS virtio 1.2 specification, §4.2 — MMIO transport register map (`MagicValue`, `Version`, `QueueNotify`). https://docs.oasis-open.org/virtio/virtio/v1.2/cs01/virtio-v1.2-cs01.html
- QEMU documentation — TCG vs. KVM, supported guest ISAs. https://www.qemu.org/docs/master/system/introduction.html

- QEMU microvm machine type — "inspired by Firecracker." https://www.qemu.org/docs/master/system/i386/microvm.html
- NEMU archived repository (archived April 14, 2021) — "Cloud Hypervisor is the successor." https://github.com/intel/nemu
- Cloud Hypervisor repository — KVM and MSHV backends, VFIO, hotplug, TDX, architecture support. https://github.com/cloud-hypervisor/cloud-hypervisor
- Cloud Hypervisor announcement, May 2019 — Intel, NEMU lineage. https://thenewstack.io/intel-releases-cloud-hypervisor-based-on-same-components-as-amazons-firecracker/
- crosvm official documentation — per-device minijail sandbox, device set, Android ARCVM. https://crosvm.dev/book/introduction.html
- crosvm rust-vmm documentation — Firecracker lineage from crosvm. https://chromium.googlesource.com/chromiumos/platform/crosvm/+/master/docs/rust-vmm.md
- rust-vmm community README — founding (December 2018), member organizations, crate inventory. https://github.com/rust-vmm/community/blob/main/README.md
- `vm-memory` crate (v0.17.1, October 2025): https://github.com/rust-vmm/vm-memory
- `kvm-ioctls` crate documentation. https://docs.rs/kvm-ioctls/latest/kvm_ioctls/
- Intel VT-x, KVM, and QEMU internals — VMX instructions, VMCS structure, EPT walkthrough. https://binarydebt.wordpress.com/2018/10/14/intel-virtualisation-how-vt-x-kvm-and-qemu-work-together/

Chapter 2: What A Virtual Machine Actually Is

Run a Linux container and a Linux VM on the same host and both feel like isolated environments: each has its own filesystem, its own process table, its own IP address. But one of them is issuing system calls directly into the host kernel's dispatch table, and the other is not. That difference is not cosmetic. It determines the threat model, the boot time, the overhead budget, and what the workload can do to the host. The boundary is drawn in hardware, at the privilege ring level the CPU enforces.

The Popek-Goldberg Theorem

In July 1974, Gerald Popek and Robert Goldberg published "Formal Requirements for Virtualizable Third Generation Architectures" in *Communications of the ACM* 17(7), pp. 412–421. The paper is short by modern standards, but it gave the field its vocabulary, and the problem it addressed was not abstract.

The question Popek and Goldberg asked was: what does a computer architecture need to guarantee so that a **Virtual Machine Monitor** (VMM) can be built for it? They wanted necessary and sufficient conditions, not a design sketch. A VMM must satisfy three properties:

- **Equivalence** (fidelity): a program running under the VMM behaves essentially identically to running on bare hardware, modulo timing and resource availability.
- **Resource control** (safety): the VMM maintains complete control over all virtualized resources. Guest code cannot directly access or modify resources the VMM has not granted.
- **Efficiency** (performance): a statistically dominant fraction of instructions executes without VMM intervention.

Efficiency is the property that constrains architecture design. If the VMM must intercept every instruction, a guest runs orders of magnitude slower than native. The only way to satisfy efficiency while preserving equivalence and control is **trap-and-emulate**: run the guest natively for most instructions and intercept only the instructions that touch privileged state.

For trap-and-emulate to work, the architecture must cooperate. Popek and Goldberg defined the relevant instruction classes. **Privileged instructions** trap when the processor is in user mode and do not trap when it is in supervisor mode. **Sensitive instructions** are those that either alter resource configuration (**control-sensitive**) or whose behavior depends on resource configuration (**behavior-sensitive**). Innocuous instructions — neither control-sensitive nor behavior-sensitive — require no VMM intervention and run at full hardware speed. The theorem follows directly:

> *"For any conventional third-generation computer, an effective VMM may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions."*

If every sensitive instruction is also privileged, then a VMM can deprivilege the guest OS — run it in a non-privileged ring — and know that any time the guest tries to touch privileged state, the CPU will trap to the VMM. The VMM inspects the instruction, emulates the intended effect within the guest's view of the machine, and returns control. Guest user-mode code never needs interception at all because it cannot access privileged state in the first place.

The 1974 paper analyzed the IBM 360, Honeywell 6000, and PDP-10. It predates x86 by several years, which turned out to matter enormously.

Why x86 Broke the Theorem

The IA-32 instruction set contains sensitive but non-privileged instructions — they execute at ring 3 or ring 1 without trapping, which means a VMM running the guest OS in a deprivileged ring cannot intercept them. The first-person account of the problem, and the definitive primary source for the count and consequences, is Bugnion, Devine, Govil, and Rosenblum, "Bringing Virtualization to the x86 Architecture with the Original VMware Workstation," ACM TOCS 2012, DOI 10.1145/2382553.2382554.

The most instructive of these is `POPF`. At ring 0, `POPF` pops a value from the stack into EFLAGS, including the Interrupt Flag (IF, bit 9) and the I/O Privilege Level field. At ring 1 with IOPL=0, the situation changes silently: CPL > IOPL, so `POPF` completes but does not modify the IF bit and does not trap. A VMM that deprivileges the guest OS kernel to ring 1 faces this consequence: a guest `POPF` that should disable interrupts — the instruction a kernel issues before a critical section — is silently swallowed. The guest OS believes interrupts are disabled. They are not. The CPU does not trap. The VMM never knows. The guest's interrupt-masking logic is now broken in a way that produces subtle, unpredictable misbehavior rather than an immediate fault.

`PUSHF` creates the mirror problem: it pushes EFLAGS to the stack and thereby reveals the real IF state to the guest. `SGDT` and `SIDT` are more aggressive still: they store the actual host GDTR and IDTR values — including the base linear addresses of the descriptor tables — to any guest-supplied memory address at any privilege level. A VMM running with real descriptor tables loaded cannot prevent a guest from reading those addresses. This violates the resource-control property directly: the VMM is not in complete control of all virtualized resources. `SLDT` stores the real LDTR; `SMSW` stores the low 16 bits of CR0. Together with instructions whose behavior varies by privilege level — `LAR`, `LSL`, `VERR`, `VERW`, segment register manipulation via `POP`, `PUSH`, and `MOV`, far calls via `CALL FAR`, `JMP FAR`, `INT n`, `RETF`, and `STR` — the full tally from the Bugnion et al. retrospective is seventeen or eighteen sensitive-but-not-privileged instructions on IA-32, depending on whether `MOV` to and from segment registers is counted as one or two distinct problematic forms.

Intel introduced `UMIP` (User-Mode Instruction Prevention, `CR4.UMIP`, bit 11) as a late partial mitigation: when enabled, `SGDT`, `SIDT`, `SLDT`, `SMSW`, and `STR` at CPL > 0 raise `#GP(0)`. UMIP appeared with Cannon Lake and Goldmont Plus (Intel) and Zen 2 (AMD). It addresses the descriptor-table leak at user

mode but does not retroactively make IA-32 virtualizable under Theorem 1 — it arrived decades after the problem was first identified and does not cover the full set.

The consequence is that IA-32 is, in Popek and Goldberg's formal sense, not virtualizable. Building a VMM for x86 required working around the architecture.

Three Approaches to an Unvirtualizable Architecture

Binary Translation

VMware's solution, first shipped as VMware Workstation in 1999, was dynamic binary translation. The VMM occupies ring 0. The guest OS kernel runs at ring 1 — a technique called **ring compression** because the full ring gap between user mode and supervisor mode is compressed into a single step. Guest applications continue at ring 3.

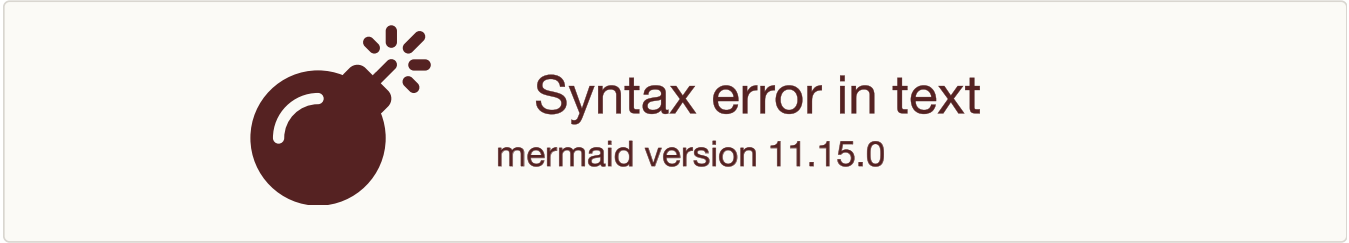
Before any basic block of guest kernel code executes, the VMM scans it for sensitive-but-not-privileged instructions and rewrites them with safe equivalents — typically calls into the VMM itself. Translated blocks are stored in a **code cache** so that frequently executed kernel paths are rewritten only once; the amortized overhead on a warm cache is small. Guest user-mode code runs natively without scanning, because user-mode code cannot issue privileged instructions even on bare hardware.

This achieves what Popek and Goldberg call **full virtualization**: the guest OS runs unmodified. The same kernel binary that boots on bare hardware boots inside the VM. The cost is translation overhead on guest kernel paths, particularly for kernel code that exercises the problematic instructions frequently.

Paravirtualization

The Xen hypervisor, presented at SOSP 2003 by Barham, Dragovic, Fraser, Hand, Harris, Ho, Neugebauer, Pratt, and Warfield, took a different position: instead of hiding the fact of virtualization, tell the guest OS it is running in a VM and let it cooperate. The hypervisor occupies ring 0. The guest OS kernel runs at ring 1. Guest applications run at ring 3.

The 17 problematic instructions are replaced in the guest OS kernel with explicit **hypercalls** — direct calls into the hypervisor's published interface. A guest that wants to update its page tables calls `__HYPERVISOR_mmu_update`. A guest that wants to set a new GDT calls `__HYPERVISOR_set_gdt`. The hypervisor validates the request and performs the operation. There is no code cache, no scanning, no rewriting. The path is direct.



The trade-off is that the guest OS must be ported. Xen shipped with modified Linux and NetBSD kernels; running an unmodified Windows guest required separate binary translation support. In Popek-Goldberg terms, paravirtualization sidesteps the sensitive-but-not-privileged problem by making the guest replace the problematic instructions itself — an approach that Popek and Goldberg's Theorem 3 licenses as a **hybrid VMM** construction.

Paravirtualization did not disappear when hardware-assisted virtualization arrived. The KVM paravirt MSR interface gives guests running under KVM access to information that hardware cannot provide natively: `MSR_KVM_STEAL_TIME` at address `0x4b564d03` reports the time a vCPU spent not scheduled on a physical core; `MSR_KVM_SYSTEM_TIME_NEW` at `0x4b564d01` provides a high-precision monotonic clock that accounts for TSC migration between cores. A guest detects KVM via `CPUID` leaf `0x40000001` — bit 3 of that leaf gates `MSR_KVM_SYSTEM_TIME_NEW` — and opts into these interfaces by writing to the MSRs. The device model is hardware-assisted; the time-keeping interface is still paravirtual.

Hardware-Assisted Virtualization

The cleanest solution was to fix the architecture. Intel shipped the first VT-x processors on **November 14, 2005** — the Pentium 4 Prescott 2M, models 662 and 672. AMD followed with AMD-V on **May 23, 2006** in Athlon 64 "Orleans" and Athlon 64 X2 "Windsor" processors. Both extensions solve the Popek-Goldberg problem by adding new CPU operating modes that make sensitive instructions either trap or become irrelevant.

VMX: The Mechanics of Hardware-Assisted Virtualization

Intel's implementation is called VMX (Virtual Machine Extensions). The CPU gains two orthogonal sub-modes, each with a full ring-0-through-ring-3 hierarchy:

- **VMX root operation**: where the hypervisor runs. Full access to all CPU state.
- **VMX non-root operation**: where guest code runs. Physically on the same silicon, but in a mode where designated instructions and events cause an automatic transfer of control to the hypervisor.

```
flowchart TB
  subgraph root["VMX Root Operation"]
    vmm["VMM / KVM (ring 0)"]
    hostuser["Host userspace (ring 3)"]
  end
  subgraph nonroot["VMX Non-Root Operation"]
    guestkernel["Guest kernel (ring 0)"]
    guestuser["Guest userspace (ring 3)"]
  end
  vmm -- "VMLAUNCH / VMRESUME" --> guestkernel
  guestkernel -- "VM exit" --> vmm
  guestuser -- "syscall" --> guestkernel
```

The transition from VMX non-root to VMX root — a **VM exit** — happens automatically when the guest executes an instruction or triggers an event configured to cause an exit. The CPU saves the guest's register state, loads the host's register state, and jumps to the VMM's exit handler. The transition in the opposite direction — a **VM entry** — happens when the VMM executes `VMLAUNCH` (on the first entry for a given vCPU) or `VMRESUME` (on every subsequent entry). The guest OS sees none of this machinery. It runs at non-root ring 0 and believes it is the most privileged software on the machine; there is no software-visible register the guest can read to discover that it is in VMX non-root mode.

The data structure that controls all of this is the **VMCS** (Virtual Machine Control Structure): a 4 KiB-aligned in-memory region, one per vCPU, whose physical address is tracked by the CPU after the hypervisor executes `VMPTRLD` on it. The VMCS has six logical groups:

Group	Purpose
Guest-state area	RSP, RIP, RFLAGS, CR0/CR3/CR4, segment registers and descriptors — saved on VM exit, loaded on VM entry
Host-state area	Host RIP, RSP, CR0/CR3/CR4, segment selectors — loaded on VM exit so the VMM resumes at a fixed handler
VM-execution control fields	Which events cause exits: `CPUID`, I/O ports, MSR reads/writes, EPT violations
VM-exit control fields	Exit-path behavior: 64-bit host, which MSRs to save/load
VM-entry control fields	Entry-path behavior: event injection, which MSRs to load
VM-exit information fields	Read-only after exit: exit reason, exit qualification, guest linear address

The VMM reads and writes VMCS fields using the `VMREAD` and `VMWRITE` instructions with 16-bit field encodings — there is no direct memory-mapped access to the VMCS bytes. The physical layout is implementation-defined by the CPU vendor and not documented. The Linux kernel's VMCS field encodings are in `arch/x86/include/asm/vmx.h`; for example, `GUEST_RIP` is `0x0000681e`, `GUEST_CR0` is `0x00006800`, `CPU_BASED_VM_EXEC_CONTROL` is `0x00004002`, and `HOST_RIP` is `0x00006c16`. The CPU also maintains a per-VMCS **launch state** — "clear" after `VMCLEAR`, "launched" after a successful `VMLAUNCH` — that is not readable via `VMREAD`. The hypervisor must use `VMLAUNCH` on the first VM entry and `VMRESUME` on all subsequent ones; the sequence is enforced by hardware.

Some instructions cause unconditional VM exits regardless of what the VMCS execution controls say: `CPUID`, `GETSEC`, `INVD`, `XSETBV`, and all VMX instructions including `VMCALL` and `VMLAUNCH` itself. Other exits are conditional, controlled by `CPU_BASED_VM_EXEC_CONTROL` and the secondary controls at `SECONDARY_VM_EXEC_CONTROL` (`0x0000401e`). The exit reasons are defined in `arch/x86/include/uapi/asm/vmx.h`, which lists 85 distinct codes. A selection:

Code	Cause
`EXIT_REASON_EXCEPTION_NMI` (0)	Guest exception or NMI
`EXIT_REASON_CPUID` (10)	Guest executed `CPUID`
`EXIT_REASON_HLT` (12)	Guest executed `HLT`
`EXIT_REASON_VMCALL` (18)	Guest issued `VMCALL`
`EXIT_REASON_IO_INSTRUCTION` (30)	Guest executed `IN` or `OUT`
`EXIT_REASON_MSR_READ` (31)	Guest executed `RDMSR`
`EXIT_REASON_MSR_WRITE` (32)	Guest executed `WRMSR`
`EXIT_REASON_EPT_VIOLATION` (48)	Guest accessed an unmapped guest-physical address
`EXIT_REASON_EPT_MISCONFIG` (49)	EPT paging structure misconfigured

Intel added 13 new instructions for the VMX interface:

Instruction	Opcode	Purpose
`VMXON m64`	`F3 0F C7 /6`	Enter VMX operation
`VMXOFF`	`NP 0F 01 C4`	Exit VMX operation
`VMPTRLD m64`	`NP 0F C7 /6`	Make a VMCS current
`VMPTRST m64`	`NP 0F C7 /7`	Store current VMCS pointer
`VMCLEAR m64`	`66 0F C7 /6`	Initialize or clear a VMCS
`VMLAUNCH`	`NP 0F 01 C2`	VM entry (first launch)
`VMRESUME`	`NP 0F 01 C3`	VM entry (resume after exit)
`VMREAD reg,r/m`	`NP 0F 78 /r`	Read a VMCS field
`VMWRITE reg,r/m`	`NP 0F 79 /r`	Write a VMCS field
`VMCALL`	`NP 0F 01 C1`	Hypercall from guest to VMM
`INVEPT reg,m128`	`66 0F 38 80 /r`	Invalidate EPT TLB entries (Nehalem+)
`INVVPID reg,m128`	`66 0F 38 81 /r`	Invalidate VPID TLB entries
`VMFUNC`	`NP 0F 01 D4`	Invoke a VM function (Haswell / Silvermont+)

Enabling VMX requires setting `CR4.VMXE` (bit 13) and then executing `VMXON` with the physical address of a 4 KiB-aligned VMXON region. Before `VMXON`, software reads the `IA32_VMX_BASIC` MSR at address `0x480` to obtain the VMCS revision identifier (bits 30:0) and writes it into bytes 0–3 of the VMXON region; a mismatch causes `VMXON` to fail. The MSR also encodes the size of both regions in bits 44:32 and the memory type for VMCS access in bits 53:50 (6 = write-back).

Memory Virtualization: The Second Translation

The VMCS and VM exits handle CPU state. Memory requires a parallel mechanism. A guest OS manages its own page tables, mapping guest-virtual addresses to what it believes are physical addresses. The host cannot use those "physical" addresses directly — the guest does not own physical DRAM in the traditional sense; it owns regions of host virtual memory that the VMM has allocated.

Intel's solution, introduced with Nehalem in 2008, is **EPT** (Extended Page Tables). EPT adds a second level of hardware page-table translation: after the guest's page tables map a guest-virtual address to a guest-physical address, the hardware's MMU automatically walks a second set of tables — maintained by the VMM but enforced by the hardware — to translate the guest-physical address to the host-physical address. Both translations happen entirely in hardware for most accesses. `INVEPT` invalidates EPT-derived TLB entries; `INVVPID` invalidates VPID TLB entries that tag TLB entries by per-VM identifier to allow host and guest TLB entries to coexist.

The `KVM_SET_USER_MEMORY_REGION` ioctl, covered in the next section, is what tells KVM where to build those EPT mappings. The guest's "physical" address `0x0` maps to whatever host-virtual address the VMM passed as `userspace_addr`. An EPT violation — exit reason `EXIT_REASON_EPT_VIOLATION` (48) — occurs when the guest accesses a guest-physical address the EPT does not yet map.

AMD's equivalent is **NPT** (Nested Page Tables, also called RVI — Rapid Virtualization Indexing), introduced with the third-generation Opteron "Barcelona" (Family 0x10). The mechanism is structurally identical to EPT; the instruction for TLB invalidation by ASID is `INVLPGA`.

KVM: The Linux Kernel's VMM

KVM (Kernel-based Virtual Machine) was developed by Avi Kivity at Qumranet and announced on October 19, 2006. It was merged into the Linux kernel in version **2.6.20**, released **February 5, 2007**. Qumranet was acquired by Red Hat in 2008. KVM ships as three loadable kernel modules: `kvm.ko` (the arch-independent core), plus either `kvm-intel.ko` or `kvm-amd.ko` depending on the CPU vendor. The arch-specific backends live in `arch/x86/kvm/vmx/vmx.c` (VMX) and `arch/x86/kvm/svm/svm.c` (SVM). KVM exposes its interface through a character device at `/dev/kvm`.

> ***Note:*** *`/dev/kvm` requires read/write access — on most Linux distributions that means membership in the `kvm` group, or root. `KVM_CREATE_VM` on a machine without hardware virtualization support will fail with `EINVAL` or `ENODEV` depending on the kernel version.*

The KVM API is organized as a three-level file descriptor hierarchy:

```
open("/dev/kvm")          → system fd
  KVM_CREATE_VM           → VM fd
    KVM_CREATE_VCPU       → vCPU fd
```

`KVM_GET_API_VERSION` on the system fd returns the constant **12** (`KVM_API_VERSION = 12`). This value has not changed since the API was frozen, and KVM documents it as a stable handshake; a caller that receives anything other than 12 has a kernel incompatibility. Every KVM ioctl uses `KVMIO = 0xAE` as the ioctl magic byte: `KVM_GET_API_VERSION = _IO(KVMIO, 0x00)`, `KVM_CREATE_VM = _IO(KVMIO, 0x01)`.

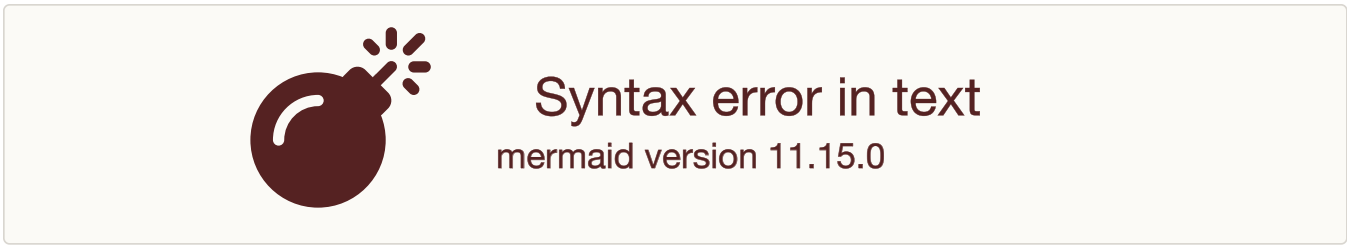
To run a guest, a VMM executes this sequence:

1. `open("/dev/kvm")` → system fd
2. `ioctl(kvm_fd, KVM_CREATE_VM, 0)` → VM fd (`machine_type = 0` for default x86)
3. `ioctl(vm_fd, KVM_SET_USER_MEMORY_REGION, &region)` — maps host `mmap`-backed pages into guest-physical space
4. `ioctl(vm_fd, KVM_CREATE_VCPU, 0)` → vCPU fd
5. `mmap(NULL, mmap_size, PROT_READ|PROT_WRITE, MAP_SHARED, vcpu_fd, 0)` → `struct kvm_run *`
6. `ioctl(vcpu_fd, KVM_RUN, 0)` — run until a VM exit KVM cannot handle internally

The struct passed to `KVM_SET_USER_MEMORY_REGION` encodes exactly how host memory becomes guest memory:

The guest's "physical" address space is a range of the VMM's virtual address space. When the guest accesses guest-physical address `0x0`, EPT translates that to the host-physical page backing `userspace_addr + 0x0`. The terminology is genuinely disorienting: the guest believes it has physical memory; the host sees a virtual address range that its own page tables back with physical DRAM. EPT is precisely the mechanism that makes both views consistent without any VMM intervention on most accesses.

After `KVM_RUN` returns, the VMM reads `kvm_run.exit_reason` from the shared page:



Selected exit reasons from `include/uapi/linux/kvm.h` (44 constants defined through `KVM_EXIT_SNP_REQ_CERTS = 43` in current mainline):

Value	Name	Cause
0	`KVM_EXIT_UNKNOWN`	Hardware exit reason not recognized by KVM
2	`KVM_EXIT_IO`	Guest executed `IN` or `OUT`
3	`KVM_EXIT_HYPERCALL`	Guest executed `VMCALL` or `VMMCALL`
5	`KVM_EXIT_HLT`	Guest executed `HLT`
6	`KVM_EXIT_MMIO`	Guest accessed an MMIO region
8	`KVM_EXIT_SHUTDOWN`	Guest triple-faulted or issued a CPU reset
9	`KVM_EXIT_FAIL_ENTRY`	Hardware refused VM entry
17	`KVM_EXIT_INTERNAL_ERROR`	KVM internal error
24	`KVM_EXIT_SYSTEM_EVENT`	Guest requested shutdown or reset
29	`KVM_EXIT_X86_RDMSR`	Guest read an MSR with no in-kernel handler
30	`KVM_EXIT_X86_WRMSR`	Guest wrote an MSR with no in-kernel handler
37	`KVM_EXIT_NOTIFY`	VM notification event
39	`KVM_EXIT_MEMORY_FAULT`	Memory access fault

When `exit_reason` is `KVM_EXIT_IO` (2), the `io` union arm carries the port number, direction, size, and `data_offset`. That last field is relative to the start of the `kvm_run` struct — the I/O data lives inline in the shared page, not in a separate buffer.

`KVM_RUN` returns to userspace only when KVM cannot handle an exit internally. KVM processes many exits entirely in the kernel: EPT violations that can be satisfied by existing mappings, in-kernel APIC accesses, in-kernel PIC and PIT emulation, and MSR reads/writes it handles itself. Only exits that genuinely require device emulation or policy decisions surface to the VMM userspace process. The performance cost of each round-trip through the VM exit path — the VMCS save/restore, the ring transition, the jump to the KVM exit handler, and the return — is non-trivial. When Spectre v2

mitigations introduced retpolines on the exit path, QEMU engineers restructured KVM's exit path to eliminate the indirect calls and recovered double-digit percentage performance improvements on exit-heavy workloads. Firecracker's answer is the smallest possible device model: only the exits that serverless workloads require.

One Kernel or Two

The mechanical difference between a container and a VM reduces to one question: how many kernel instances are running?

A **container** is a process or process tree whose system calls go directly into the host kernel's dispatch table. Linux namespaces partition the kernel's exported interfaces into isolated views — one set for mount points, another for the PID space, another for the network stack — and cgroups constrain the resources each group can consume. But no namespace or cgroup puts silicon between a container process and the kernel's system-call table. Every container on a host shares one running kernel, differentiated only by namespace membership. There is no second OS image. A vulnerability reachable via any system call available inside the container is a host-wide concern.

A **VM** carries a complete second kernel binary. That binary — on x86-64, a compressed `bzImage` following the x86 Linux boot protocol documented in `Documentation/arch/x86/boot.rst` — is loaded by the VMM into guest-physical memory, its entry point is written into the VMCS guest-state area at `GUEST_RIP`, and `VMLAUNCH` transfers control. The guest kernel boots, builds its own process table, installs its own interrupt handlers, manages its own memory map, and runs its own init. When a process inside the VM issues a system call, the syscall enters the guest kernel at non-root ring 0. The host kernel never sees it as a system call. The host sees only VM exits from the VMCS — I/O port accesses, MMIO accesses, MSR reads — which the VMM handles in userspace before re-entering the guest. The guest's syscall dispatch table, its kernel configuration, its module set, and its kernel CVE surface are all independent of the host.

```
flowchart TB
  subgraph container["Container (one kernel)"]
    cp["Container process"]
    hk["Host kernel syscall table"]
    cp -- "syscall" --> hk
  end
  subgraph vm["VM (two kernels)"]
    gp["Guest process"]
    gk["Guest kernel (bzImage)"]
    vmm["VMM + KVM"]
    hk2["Host kernel"]
    gp -- "syscall" --> gk
    gk -- "VM exit (I/O, MMIO, MSR)" --> vmm
    vmm -- "ioctl(vcpu_fd, KVM_RUN)" --> hk2
  end
```

A guest kernel crash surfaces as `KVM_EXIT_SHUTDOWN` (value 8) in the host VMM's `kvm_run.exit_reason`; the VMM tears down the VM, and the host kernel and all other VMs are unaffected. Escaping the VM requires exploiting the VMM or a hardware flaw in the virtualization extensions, not a guest kernel CVE. On the other side of the ledger: boot time is non-trivial even for a stripped guest kernel — tens of milliseconds to reach the first userspace process before counting VMM startup — and every VM carries its own kernel memory overhead, on the order of tens of MiB for a minimal configuration.

A container's startup time is proportional to `clone(2)` plus `execve(2)`, measured in milliseconds, with no kernel boot. The isolation boundary is the host kernel's software enforcement of namespaces and cgroups, not a hardware mode transition.

That means understanding, and then ruthlessly minimizing, what a VMM is actually required to do.

Native Execution and Emulation Inside a VM

Within a running VM, two execution regimes alternate. Most of the time the guest runs in **native execution**: the CPU is in VMX non-root mode, guest ring 0 and ring 3 code executes directly on physical silicon at full hardware speed, and neither KVM nor the VMM interprets any instructions. This is what the efficiency property in Popek and Goldberg's theorem demands, and hardware-assisted virtualization delivers it: most guest instructions never involve the VMM at all.

Emulation happens when a VM exit occurs and the VMM must act before re-entering the guest. When the guest's kernel driver issues an `OUT` instruction to a port that maps to a virtual UART, the hardware exits to VMX root mode and KVM delivers `KVM_EXIT_IO` (value 2) to the VMM userspace process. The VMM reads the port, direction, and size from `kvm_run.io`, performs whatever logic the device model requires — in `firecracker`'s case, forwarding bytes to the 16550A serial emulation — and calls `ioctl(vcpu_fd, KVM_RUN, 0)` again. The guest never knows the UART is a data structure in the VMM's heap.

```
sequenceDiagram
  participant G as "Guest (VMX non-root)"
  participant K as "KVM (kernel)"
  participant V as "VMM userspace (firecracker)"

  G->>K: OUT to serial port (VM exit)
  K->>K: Check exit reason
  alt KVM handles in-kernel
    K->>G: VMRESUME
  else requires userspace
    K->>V: KVM_RUN returns, exit_reason = KVM_EXIT_IO
    V->>V: 16550A device model
    V->>K: ioctl(vcpu_fd, KVM_RUN, 0)
    K->>G: VMRESUME
  end
```

EPT violations that KVM can resolve — a first access to a mapped but not yet faulted-in page — are handled in the page-fault path without the VMM process being involved. Each exit that does surface to userspace carries the full cost of the VMM round-trip: VMCS save/restore, ring transition, exit handler dispatch, and re-entry. `KVM_RUN` returns to userspace only for exits the VMM must handle itself.

The exits that do reach userspace are the ones that constitute the device model. A traditional VMM like `qemu-system-x86_64` emulates dozens of virtual devices, each with its own I/O port ranges and MMIO regions; every device access by the guest is a VM exit, a round-trip through the exit path, and a return to the guest. Firecracker reduces this to the smallest device set that serverless workloads require — virtio block, virtio net, virtio vsock, virtio rng, a 16550A serial console, an i8042 keyboard controller stub, and a balloon device — and devices that are not present cannot generate exits. The device count is a directly tunable multiplier on VMM overhead. Chapter 3 returns to what that overhead actually costs.

Sources And Further Reading

- Popek, G. J., and Goldberg, R. P., "Formal Requirements for Virtualizable Third Generation Architectures," *Communications of the ACM* 17(7), pp. 412–421, July 1974. The paper establishing the three VMM properties and the trap-and-emulate theorem. https://dl.acm.org/doi/10.1145/361011.361073
- Wikipedia treatment of the Popek-Goldberg requirements with original citation and theorem text: https://en.wikipedia.org/wiki/Popek_and_Goldberg_virtualization_requirements
- "The Morning Paper" summary of the Popek-Goldberg paper: https://blog.acolyer.org/2016/02/19/formal-requirements-for-virtualizable-third-generation-architectures/
- Bugnion, E., Devine, S., Govil, K., and Rosenblum, M., "Bringing Virtualization to the x86 Architecture with the Original VMware Workstation," *ACM Transactions on Computer Systems*, 2012. The

definitive primary source on binary translation and the 17 sensitive-but-not-privileged IA-32 instructions. https://dl.acm.org/doi/pdf/10.1145/2382553.2382554

- Barham, P., et al., "Xen and the Art of Virtualization," ACM SOSP 2003. Introduces paravirtualization and the hypercall model. https://www.cl.cam.ac.uk/research/srg/netos/papers/2003-xensosp.pdf
- Xen hypercall table reference (`__HYPERVISOR_*` constants from `xen.h`): https://xenbits.xen.org/docs/unstable/hypercall/x86_64/include,public,xen.h.html
- `POPF` / `PUSHF` behavior by privilege level: https://www.felixcloutier.com/x86/popf:popfd:popfq
- LWN article on UMIP (`CR4.UMIP`): https://lwn.net/Articles/721957/
- `SGDT` / `SIDT` information-leak discussion: http://hypervsir.blogspot.com/2014/10/kernel-information-leak-with.html
- x86 virtualization history (VT-x ship date November 14, 2005; AMD-V ship date May 23, 2006): https://en.wikipedia.org/wiki/X86_virtualization
- Intel SDM Volume 3C (VMX): https://cdrdv2-public.intel.com/671506/326019-sdm-vol-3c.pdf
- `VMLAUNCH` / `VMRESUME` instruction reference: https://www.felixcloutier.com/x86/vmlaunch:vmresume
- `IA32_VMX_BASIC` MSR field layout (revision identifier, region size, memory type): https://scrapbox.io/vmm/IA32_VMX_BASIC
- VMX instruction opcode table: https://en.wikipedia.org/wiki/List_of_x86_virtualization_instructions
- Linux kernel VMCS field encodings (`arch/x86/include/asm/vmx.h`): https://github.com/torvalds/linux/blob/master/arch/x86/include/asm/vmx.h
- Linux kernel VMX exit reason codes (`arch/x86/include/uapi/asm/vmx.h`): https://github.com/torvalds/linux/blob/master/arch/x86/include/uapi/asm/vmx.h
- AMD APM Volume 2 (doc 24593) — the authoritative SVM / VMCB reference: https://www.amd.com/system/files/TechDocs/24593.pdf
- Second Level Address Translation (EPT / NPT): https://en.wikipedia.org/wiki/Second_Level_Address_Translation
- Avi Kivity, "KVM: the Linux Virtual Machine Monitor," OLS 2007: https://www.kernel.org/doc/ols/2007/ols2007v1-pages-225-230.pdf
- KVM merged into Linux 2.6.20 — LWN history: https://lwn.net/Articles/705160/
- KVM API documentation (canonical): https://docs.kernel.org/virt/kvm/api.html
- "Using the KVM API," LWN.net — walkthrough of the three-level fd hierarchy with annotated C code: https://lwn.net/Articles/658511/
- Linux UAPI `kvm.h` (exit reason values, `kvm_run` struct): https://github.com/torvalds/linux/blob/master/include/uapi/linux/kvm.h
- KVM paravirt MSR documentation (`MSR_KVM_STEAL_TIME` , `MSR_KVM_SYSTEM_TIME_NEW` , `CPUID` leaf `0x40000001`): https://www.kernel.org/doc/html/latest/virt/kvm/x86/msr.html
- QEMU blog — micro-optimizing KVM VM exits (Spectre retpoline overhead): https://www.qemu.org/2019/11/15/micro-optimizing-kvm-vmexits/

- Linux x86 boot protocol (`Documentation/arch/x86/boot.rst`): https://www.kernel.org/doc/html/latest/arch/x86/boot.html
- Firecracker design document: https://github.com/firecracker-microvm/firecracker/blob/main/docs/design.md
- OSDev VMX reference: https://wiki.osdev.org/VMX

Chapter 3: Why MicroVMs Exist

The serverless billing model promises that a user pays for exactly the CPU cycles their function consumed, nothing more. Delivering that promise at scale means packing thousands of independent workloads onto a single host while giving each one a security boundary it cannot escape. Containers are the obvious first answer, and for most uses they are sufficient. But in a multi-tenant serverless platform where every customer's code runs without review on shared hardware, the container model has a structural weakness: every tenant shares one kernel. That one fact, and the effort to work around it without sacrificing density, is where microVMs come from.

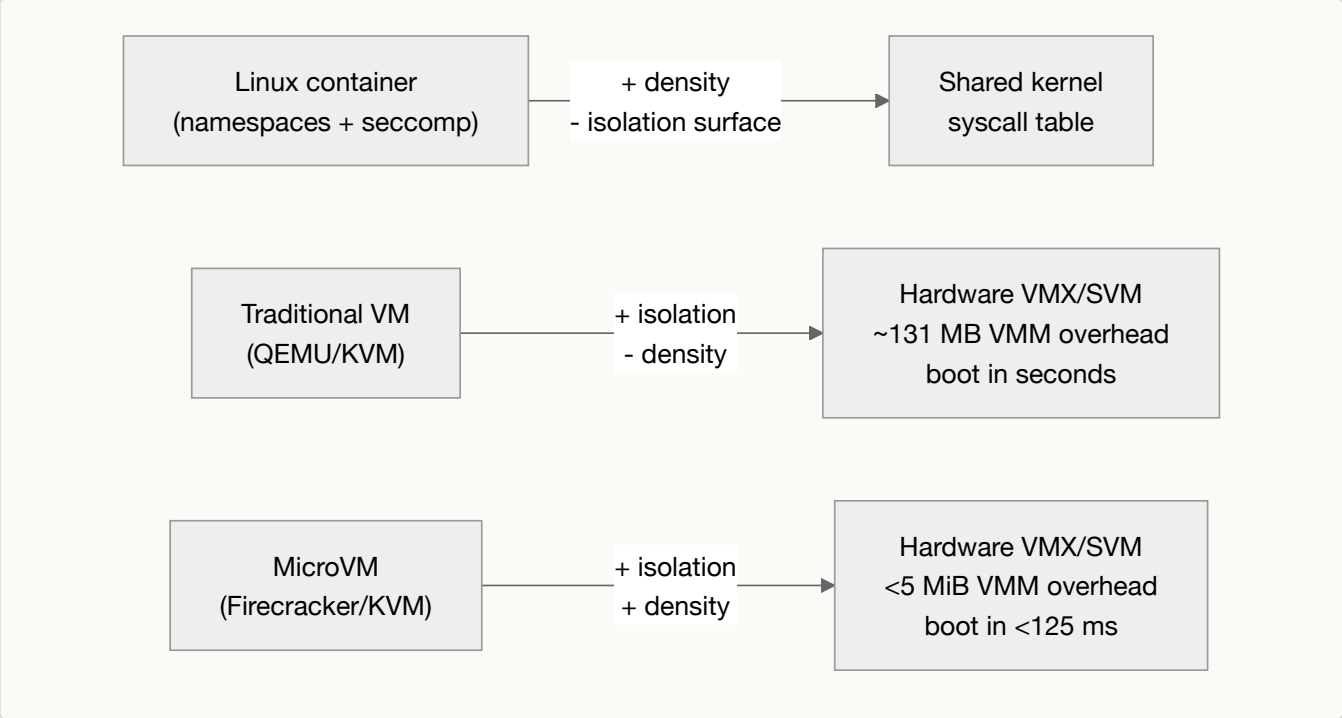
The Isolation-vs-Density Tradeoff

A Linux container is not a separate operating system instance. It is a collection of views over a shared kernel, carved out by `clone(2)`, `unshare(2)`, and `setns(2)`. Eight namespace types govern these views: mount tables (since Linux 3.8), PIDs (3.8), UTS hostname (3.0), IPC and POSIX message queues (implemented in Linux 2.6.19; the `/proc/[pid]/ns/ipc` symlink appeared in 3.0), network stacks (implemented in Linux 2.6.24; the `/proc/[pid]/ns/net` symlink appeared in 3.0), user and group IDs (3.8), cgroup roots (4.6), and monotonic and boot-time offsets (5.6). Everything inside those namespaces looks isolated. The kernel itself does not.

x86-64 Linux exposes more than 400 system calls. A process in any container reaches the same dispatch table as a process on the bare host. Any kernel vulnerability reachable through that interface is reachable from any container on the machine. Two CVEs make this concrete. CVE-2019-5736, published on 2019-02-11 with a CVSS score of 8.6, allowed a process running as uid 0 inside a container to race a file-descriptor open on `/proc/self/exe` and overwrite the host `runc` binary, achieving root code execution on the host. The `/proc` filesystem crosses namespace boundaries by design; the exploit required nothing more exotic than that. It was fixed in runc 1.0-rc7. CVE-2024-21626 ("Leaky Vessels"), disclosed in January 2024 with the same CVSS score of 8.6, exploited a file-descriptor leak in runc v1.0.0-rc93 through v1.1.11: a container process could obtain a working-directory file descriptor that pointed into the host filesystem namespace, then escape the container via `runc run` or `runc exec`. Fixed in runc 1.1.12. Both exploits share one root cause: the security-critical interface is the host kernel's shared syscall table. No namespace can close that gap — the gap is structural. Seccomp filters help, but they reduce the attack surface; they do not eliminate the shared kernel.

Traditional VMs address this differently. Intel VMX introduces root and non-root operation, enforced in silicon through the Virtual Machine Control Structure (VMCS). The guest executes entirely in non-root mode; any sensitive instruction or external event causes a VM exit that transfers control to the hypervisor in root mode. The security-critical boundary is not a software dispatch table but a hardware mode transition. An attacker inside the guest has to escape through that hardware boundary, not through a shared dispatch table. That is a fundamentally different threat model.

The cost is density. A conventional VM running under QEMU 4.2 carries roughly 131 MB of VMM process overhead — measured as the non-shared VMM process memory minus configured guest RAM on an m5d.metal instance (2x Intel Xeon Platinum 8175M, 384 GB RAM, Ubuntu 18.04.2, kernel 4.15.0-1044-aws, per NSDI 2020 Figure 7). It emulates more than 40 virtual devices, drives a BIOS, runs a bootloader, and takes seconds to reach the first userspace process. At 131 MB of overhead per VM, a 384 GB host can support roughly 2,900 fully-sized VMs before touching guest RAM at all. If functions are as small as 128 MB — the minimum Lambda allocation — the VMM overhead alone represents a full function's worth of RAM on every slot.



The AWS Lambda Origin

Firecracker was built at Amazon to solve this problem for Lambda and Fargate. The architecture that preceded it used Linux containers to isolate individual functions within a customer's account, and separate EC2 VMs to isolate between customers. That two-tier arrangement imposed a structural inefficiency: each VM had to be sized before anyone knew which tenant mix would fill it. A VM provisioned for 128 MB functions wastes capacity if larger functions land on it; a VM sized for 1.5 GB functions underserves 128 MB slots. The outer VM boundary also meant that Lambda's per-function billing had a real VM lurking behind it, with real boot latency and real idle overhead.

Lambda's production constraints dictated the design requirements precisely. A host with 1 TB of RAM running 128 MB functions needs up to 8,000 function slots to be fully packed; VMM overhead must therefore be negligible as a fraction of guest RAM, not a flat cost per slot. Lambda pre-boots a pool of

MicroVMs to absorb burst traffic — by Little's Law, at a 125 ms creation time, one pooled slot is consumed per 8 new invocations per second that arrive faster than the pool refills. Each slot lives at most 12 hours, then is recycled; the same slot handles many serial invocations of the same function.

Firecracker entered internal Lambda production in 2018. The open-source release was announced on 2018-11-27 on the AWS open-source blog under the Apache 2.0 license. The NSDI 2020 paper reports that Firecracker handles "trillions of requests per month" across Lambda and Fargate (§1, §4.1). The paper's authors are Alexandru Agache and colleagues at Amazon; it appeared at USENIX NSDI 2020.

Firecracker's Six Design Goals

The NSDI 2020 paper states six explicit design criteria in §2. They define the axes along which the paper's benchmarks are organized and against which individual design choices are justified.

Isolation. The guest boundary must be enforced by the hardware VMX/SVM mechanism, not by software policy. Every vCPU thread is treated as potentially hostile from its first instruction. In practice, this means `jailer` — Firecracker's companion process — drops into a chroot, sets up Linux namespaces, installs its own seccomp-BPF filter covering exactly 24 syscalls (with argument filtering) and 30 ioctls, and then `exec`s `firecracker`. Of those 30 ioctls, 22 are required by KVM's own ioctl-based API (`KVM_CREATE_VM`, `KVM_SET_USER_MEMORY_REGION`, `KVM_RUN`, and so on). The 24/30 figures are the jailer's own filter; Firecracker itself runs three thread types — API server, VMM, and vCPU threads — each with a separate, narrower seccomp-BPF filter compiled at build time from JSON by `seccompiler-bin` and embedded in the binary. A syscall not in the relevant thread's whitelist delivers `SIGSYS` to the offending thread; it does not reach the kernel at all.

Overhead and density. SPECIFICATION.md requires that the Firecracker VMM process consume no more than 5 MiB of overhead — defined as non-shared VMM process memory minus configured guest RAM — for a single-vCPU guest with 128 MiB of RAM using the Firecracker-tuned kernel. In practice the measured overhead is approximately 3 MiB (NSDI 2020 Figure 7). This overhead does not scale with VM size; adding RAM to the guest adds nothing to the VMM footprint. In production Lambda and Fargate, Firecracker accounts for 3% of total RAM (§5.4), compared to the 131 MB QEMU would require for each slot.

Performance. The guest compute floor, defined in SPECIFICATION.md, targets better than 95% of bare-metal throughput. For I/O the paper reports that 4 kB read P99 latency inside a Firecracker guest adds only 49 µs over the native NVMe result on the same m5d.metal host. Network throughput is specified in SPECIFICATION.md as at least 14.5 Gbps at 80% CPU utilization, or 25 Gbps at 100% CPU. These targets govern the design of the virtio device emulators discussed in later chapters.

Soft allocation. Lambda and Fargate require memory and CPU oversubscription. SPECIFICATION.md and the paper specify that Firecracker must support oversubscription; the paper evaluates it at ratios exceeding 20x and reports production operation at up to 10x (§5.4). This goal has a direct consequence for the device model, discussed below.

Fast switching. A slot must be created fast enough that the pre-boot pool can be refilled as quickly as it drains. SPECIFICATION.md defines the boot time target as the wall-clock duration from the `InstanceStart` API call until the guest kernel forks `/sbin/init`, using a minimal init, no serial console, no networking, and a minimal kernel configuration. That target is under 125 ms. SPECIFICATION.md also specifies that the Firecracker process itself — the VMM before any guest is configured — must have its API socket ready within 8 CPU-milliseconds of process start; in practice the wall-clock time is 6–60 ms, most commonly around 12 ms. The design point in `docs/design.md` is 5 MicroVMs per host core per second, the only confirmed throughput figure in primary sources; 150 MicroVMs per second per host is a commonly cited headline figure that appears on the Firecracker homepage but is not stated in SPECIFICATION.md or design.md.

Compatibility. Firecracker must run an unmodified Linux guest kernel and standard ELF binaries. No kernel patches. No guest agent. This is what makes Lambda transparently support existing code without recompilation.

Why Rust

Firecracker began life as a fork of Google's crosvm, the ChromeOS VMM, which is itself written in Rust. The team deleted USB support, GPU passthrough, the 9p filesystem driver, and other components. At the time of the NSDI 2020 paper, Firecracker contained approximately 50,000 lines of Rust — the team had added more than 20,000 new lines and changed 30,000 lines since the fork, and the codebase was fewer than half the size of crosvm at that point (NSDI 2020 §2.1). QEMU 4.2, for comparison, has more than 1.4 million lines of C.

That ratio is not a vanity metric. A smaller codebase means a smaller set of syscalls needed from the host kernel. QEMU requires up to 270 unique syscalls during operation; KVM itself adds approximately 120,000 lines of kernel code to the host's trusted computing base (paper §2.1.3). Firecracker's jailer seccomp filter reduces the host kernel's attack surface from those 270 to 24. Rust was chosen specifically because the device emulators, which handle attacker-controlled input from the guest virtio queues, must be memory-safe without relying on a garbage collector. A memory allocator pause in a VMM thread stalls a vCPU; that is unacceptable in a sub-millisecond latency path. The compile-time invariants enforced by Rust's borrow checker — in particular, the prohibition on data races across thread boundaries — are enforced at build time, not at runtime, which means they do not add runtime overhead to the critical path.

What You Give Up

Everything above is bought with deliberate omissions. Firecracker is fast and dense because it refused to implement what Lambda and Fargate do not need.

The Minimal Device Model

At the time of the NSDI 2020 paper, Firecracker emulated exactly four device types: `virtio-net` (network), `virtio-block` (storage), a 16550A-compatible serial console (`ttyS0`), and a partial i8042 keyboard controller implemented in fewer than 50 lines of Rust, used only to signal reboot and shutdown. Since then the source tree has grown to include `virtio-balloon` (`VIRTIO_ID_BALLOON = 5`), `virtio-vsock` (`VIRTIO_ID_VSOCK = 19`), `virtio-rng` (`VIRTIO_ID_RNG = 4`), `virtio-pmem` (`VIRTIO_ID_PMEM = 27`), and `virtio-mem` (`VIRTIO_ID_MEM = 24`). The FAQ lists six devices; the current source tree may diverge from that count as development continues.

All of these use the virtio MMIO transport (virtio spec §4.2), not PCI (§4.1). Each device occupies a 4 KiB window in guest physical address space; the first device is mapped at `0xc0001000`, the second at `0xc0002000`, and so on. The guest kernel learns about each device through its command line, which includes a parameter of the form `virtio_mmio.device=4K@0xc0001000:5`, where the components are `size@gpa:IRQ`. There is no PCI bus, no config space, no MMIO discovery scan — the guest is told exactly where each device lives before it boots. This eliminates the probe timeouts that add up to 900 ms to an Ubuntu 18.04 kernel boot on the same hardware.

What is absent relative to QEMU's more than 40 emulated devices: GPU (`VIRTIO_ID_GPU = 16`), virtio-console (3), SCSI (8), virtio-input (18), crypto (20), sound (25), virtiofs (26), and all physical device passthrough. These gaps are features for Lambda; they are missing dependencies for workloads that need them.

No BIOS, No PCI, No USB

Firecracker does not present a BIOS. It does not emulate legacy ISA or PCI devices. It does not support USB. The VMM loads the Linux kernel directly via the x86 Linux boot protocol, identified by the four-byte magic `0x53726448` — the ASCII string `"HdrS"` — at offset 0x202 in the kernel image. On x86-64 the guest image must be an uncompressed ELF (`vmlinux`); on aarch64 it must be a PE-formatted `Image`. The kernel command line includes `pci=off`, suppressing PCI bus enumeration entirely. There is no ACPI power management (no S5 shutdown path) on x86, and guest reboot is not supported on any architecture (FAQ.md). Firecracker does generate ACPI DSDT tables for virtio-MMIO device enumeration on recent versions — which is why `CONFIG_ACPI=y` is still required for block-device boot on x86-64, as the kernel needs ACPI to parse the device table even though it will never send an S5 event.

This also means Firecracker cannot boot an arbitrary kernel binary from a disk image — not because of a policy restriction but because the entire guest setup assumes direct kernel loading. There is no stage to hand off to GRUB.

A Curated Guest Kernel

The compatibility goal says "unmodified guest binaries," and that is true. The kernel is a different matter. SPECIFICATION.md specifies a minimum supported guest kernel of Linux 6.1 (requiring Firecracker v1.9.0 or later; support ends 2026-09-02). Linux 5.10 reached its minimum support commitment in

Firecracker on 2024-01-31 (kernel-policy.md).

For a block-device boot on x86-64, SPECIFICATION.md requires the following kernel configuration: `CONFIG_VIRTIO_MMIO=y`, `CONFIG_VIRTIO_BLK=y`, `CONFIG_ACPI=y`, `CONFIG_PCI=y`, and `CONFIG_KVM_GUEST=y`. For initrd-only boot, the last three can be omitted. The `CONFIG_KVM_GUEST=y` flag enables `CONFIG_KVM_CLOCK`, which replaces the TSC and HPET-based timekeeping with paravirtualized clock reads, keeping the guest's sense of time accurate without costly VM exits on every `rdtsc`. On aarch64 the equivalent requirements are `CONFIG_ARM_AMBA=y` and `CONFIG_RTC_DRV_PL031=y`.

The reference CI kernel configuration for x86-64 Linux 6.1 (available at `resources/guest_configs/microvm-kernel-ci-x86_64-6.1.config` in the Firecracker repository) disables all physical hardware drivers: `CONFIG_USB`, `CONFIG_WIRELESS`, `CONFIG_BLUETOOTH`, `CONFIG_SOUND`, `CONFIG_DRM`, `CONFIG_FB`, `CONFIG_BLK_DEV_NVME`, `CONFIG_ATA`, `CONFIG_MD`, and `CONFIG_BLK_DEV_SD` are all unset. The compressed kernel image is 4.0 MB with no modules; Ubuntu 18.04's kernel is 6.7 MB plus 44 MB of modules.

The GPU Question

VFIO/PCI passthrough — the standard Linux mechanism for assigning a physical PCI device to a guest — is not implemented. Firecracker GitHub issue #849, opened in January 2019, tracked the feature request; Discussion #4845 (February 2025) announced that the team was pausing the effort due to insufficient internal resources. The underlying conflict is structural: VFIO requires guest memory to be pinned (non-swappable by the host) for the duration of DMA operations. Pinned memory cannot be balloon-reclaimed or tracked for dirty-page migration. That directly contradicts the soft-allocation design goal, which depends on the balloon driver to reclaim idle guest memory across thousands of slots. A future implementation would have to choose one or the other; Firecracker has chosen soft allocation.

Discussion #4845 also describes a planned MVP for GPU support that explicitly excluded multiple GPUs, peer-to-peer GPU transfers, VF passthrough, GPU snapshot/resume, GPU-Direct, NVMe, and hotplugging. That scope description is itself a summary of what the minimal device model considers out of scope. For GPU compute workloads, a container or a traditional VM is the right tool.

The I/O Ceiling

The serial I/O path — one virtio queue, one worker thread, one host file-descriptor per device — is simple and predictable, but it is not the fastest possible path. On the m5d.metal evaluation host, bare-metal NVMe sustains more than 340,000 4 kB read IOPS (approximately 1 GB/s). Inside a Firecracker guest with virtio-block, the ceiling is approximately 13,000 IOPS (52 MB/s at 4 kB). The P99 latency penalty over native is only 49 µs; the throughput gap is the cost of serial queue handling (NSDI 2020 §5.3).

Network throughput is bounded similarly. The host `tap` interface on m5d.metal reaches 44–46 Gb/s. A Firecracker guest peaks at approximately 15 Gb/s across all tested configurations (single stream, 10 concurrent streams). SPECIFICATION.md specifies a floor of 14.5 Gbps at 80% CPU or 25 Gbps at 100%

CPU — considerably below what the hardware can deliver, but also considerably above what a serverless function invocation requires. The throughput scales linearly to 16 concurrent MicroVMs, which is the production-relevant question: aggregate host bandwidth degrades gracefully as slot count increases.

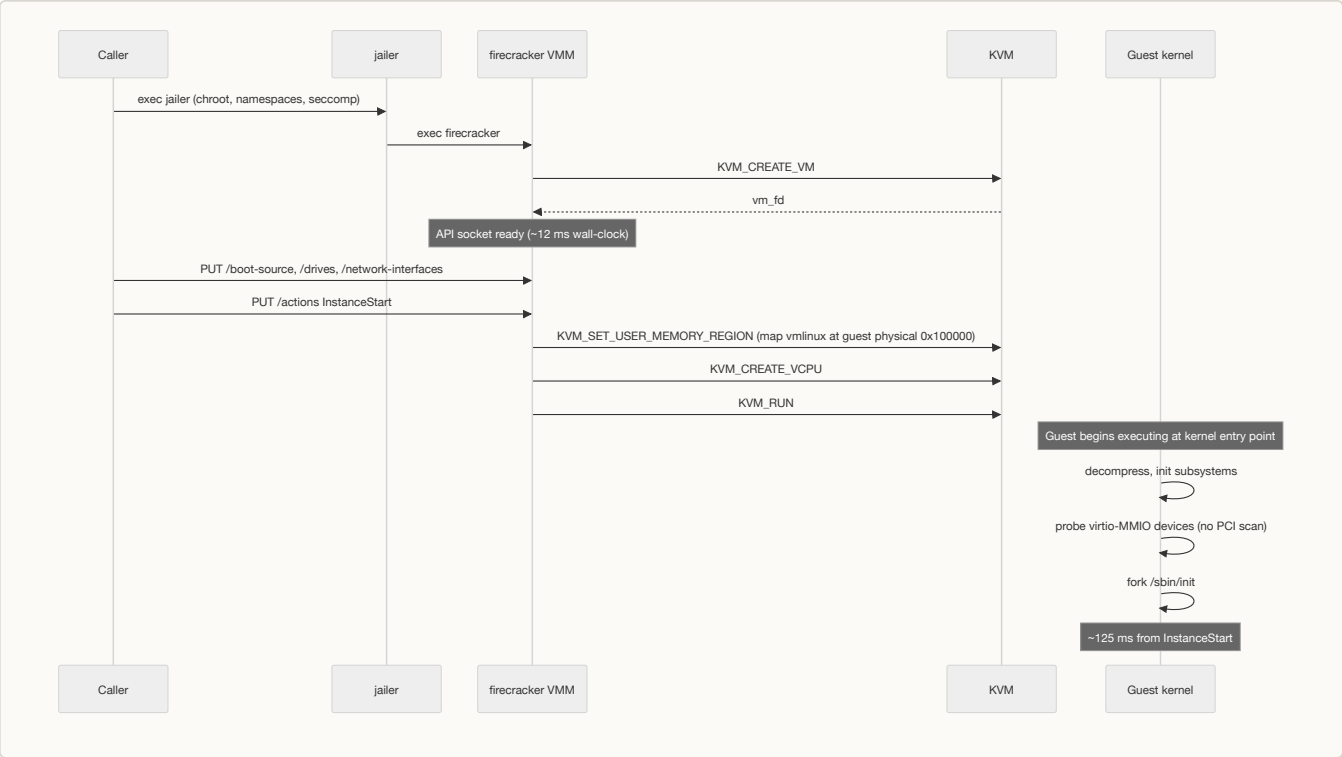
The virtio-PCI transport, which Firecracker does not support, would reduce per-operation overhead by replacing MMIO window polling with MSI-X interrupts — a structural advantage for latency-sensitive workloads. That is a separate line of work from device passthrough, and it would require adding PCI config space emulation to the VMM.

The Boot Time in Detail

The 125 ms target measures wall-clock time from `InstanceStart` to the first instruction of `/sbin/init`, under three conditions: serial console disabled, Firecracker-tuned kernel, minimal root filesystem.

Five hundred serial-boot samples on m5d.metal (NSDI 2020 §5.1) show that Firecracker in pre-configured mode ("FC-pre," where the API calls to set up the kernel and rootfs are completed before the `InstanceStart` call) boots about twice as fast as a comparable QEMU configuration. The dominant cost in a Linux kernel boot on that hardware is not CPU work — it is device probe timeouts. An Ubuntu 18.04 kernel adds approximately 900 ms compared to a minimal Firecracker kernel because it probes for hardware that does not exist; those probes time out rather than fail fast. Disabling the serial console (`console=` absent from the kernel command line) saves up to 70 ms. Adding a single statically configured network interface costs 20 ms (Firecracker) or 35 ms (QEMU).

In parallel-boot testing — 1,000 MicroVMs with 50 concurrent creation requests — FC-pre achieves a P99 boot time of 146 ms. At 100 concurrent: 153 ms. The 125 ms target is a serial-path SLA; the parallel path adds contention for KVM device nodes and host memory bandwidth. This is why Lambda keeps a pre-booted pool rather than creating VMs inline with incoming requests: 125 ms is fast for a VM, and it is too slow to absorb a cold burst without buffering.



Sources And Further Reading

- Agache, A. et al., "Firecracker: Lightweight Virtualization for Serverless Applications," USENIX NSDI 2020. PDF: https://www.usenix.org/system/files/nsdi20-paper-agache.pdf
- Firecracker `SPECIFICATION.md`: https://github.com/firecracker-microvm/firecracker/blob/main/SPECIFICATION.md
- Firecracker `docs/design.md`: https://github.com/firecracker-microvm/firecracker/blob/main/docs/design.md
- Firecracker `FAQ.md`: https://github.com/firecracker-microvm/firecracker/blob/main/FAQ.md
- Firecracker `docs/kernel-policy.md`: https://github.com/firecracker-microvm/firecracker/blob/main/docs/kernel-policy.md
- Firecracker reference kernel config (x86-64, Linux 6.1): https://github.com/firecracker-microvm/firecracker/blob/main/resources/guest_configs/microvm-kernel-ci-x86_64-6.1.config
- Firecracker GPU/VFIO Discussion #4845: https://github.com/firecracker-microvm/firecracker/discussions/4845
- Firecracker GPU Issue #849: https://github.com/firecracker-microvm/firecracker/issues/849
- AWS open-source launch blog (2018-11-27): https://aws.amazon.com/blogs/opensource/firecracker-open-source-secure-fast-microvm-serverless/
- OASIS virtio 1.2 specification (CS01), §4.2 (MMIO transport): https://docs.oasis-open.org/virtio/virtio/v1.2/cs01/virtio-v1.2-cs01.html
- Linux KVM API documentation: https://www.kernel.org/doc/html/latest/virt/kvm/api.html

- Linux x86 boot protocol: https://www.kernel.org/doc/html/latest/arch/x86/boot.html
- Linux `namespaces(7)` man page: https://man7.org/linux/man-pages/man7/namespaces.7.html
- CVE-2019-5736 (runc container escape, CVSS 8.6): https://nvd.nist.gov/vuln/detail/CVE-2019-5736
- CVE-2024-21626 ("Leaky Vessels", runc file-descriptor escape, CVSS 8.6): https://nvd.nist.gov/vuln/detail/CVE-2024-21626
- `linux/virtio_ids.h`: https://raw.githubusercontent.com/torvalds/linux/master/include/uapi/linux/virtio_ids.h
- `linux/kvm.h`: https://raw.githubusercontent.com/torvalds/linux/master/include/uapi/linux/kvm.h
- Firecracker NSDI 2020 benchmark data: https://github.com/firecracker-microvm/nsdi2020-data

PART II — HARDWARE AND KERNEL PRIMITIVES

Chapter 4: CPU Virtualization Extensions

Before hardware added explicit virtualization support, a VMM had to catch every privileged guest instruction by running the guest in user mode and trapping each fault. The approach was called "trap and emulate," and it worked well enough for most instructions — but x86 had a category of instructions that were sensitive without being privileged. Instructions like `SGDT`, `SIDT`, and `PUSHF` read or modify state that differs between the host and guest, yet they execute silently at ring 3 instead of faulting. A guest OS issuing `SGDT` to read the GDT register would get the host's descriptor-table base, not its own. There was no trap, so there was nothing to catch. VMware's early x86 hypervisors solved the problem with binary translation — a JIT compiler that rewrote guest code on the fly before execution, patching out the troublesome instructions. It worked, but it required a scanning pass over every basic block — and VMware's own published benchmarks showed 5–20% overhead on kernel-intensive workloads.

Intel's answer, published in 2005 as **VT-x** (Virtualization Technology for IA-32/IA-64/x86_64), and AMD's concurrent answer as **AMD-V** (also called **SVM**, Secure Virtual Machine), both solve the same problem the same way: add a hardware-managed execution mode where the CPU can run guest ring-0 code natively at full speed, intercept exactly the operations the hypervisor cares about, and save and restore the world in a processor-managed data structure. ARM took a structurally different route, adding a dedicated privilege level — EL2, higher than the OS's EL1 — that sits above the guest and controls what the guest can see. All three mechanisms share the same goal: make guest ring 0 architecturally distinct from host ring 0, enforced by hardware decode logic rather than software convention.

Intel VT-x

Detecting and Entering VMX Mode

The first thing any VMM must do is confirm the CPU supports VMX. That check is `CPUID` leaf `1`, ECX bit 5: `CPUID.01H:ECX.VMX[bit 5] = 1`. A zero there means no VT-x, full stop.

Three setup steps must happen before `VMXON` will succeed. First, the VMM sets `CR4.VMXE` (bit 13). Executing `VMXON` with that bit clear raises `#UD`, an invalid-opcode fault — the instruction does not even exist to the CPU in that state. Second, the VMM reads `IA32_FEATURE_CONTROL` (MSR address `0x3A`) and verifies bit 0 (the lock bit) and bit 2 are both set. Bit 0, once written to 1, is latched until power-on reset; BIOS programs it during POST. Bit 2 authorizes `VMXON` outside SMX operation, which is the normal case. If the lock bit is clear, firmware has not committed the machine to VMX operation and `VMXON` will fault with `#GP(0)`. Third, the VMM allocates a 4 KiB-aligned **VMXON region**, writes the 31-bit VMCS revision identifier (from `IA32_VMX_BASIC` MSR `0x480`, bits 30:0) into its first four bytes with bit 31 cleared, and passes its physical address as the operand to `VMXON`.

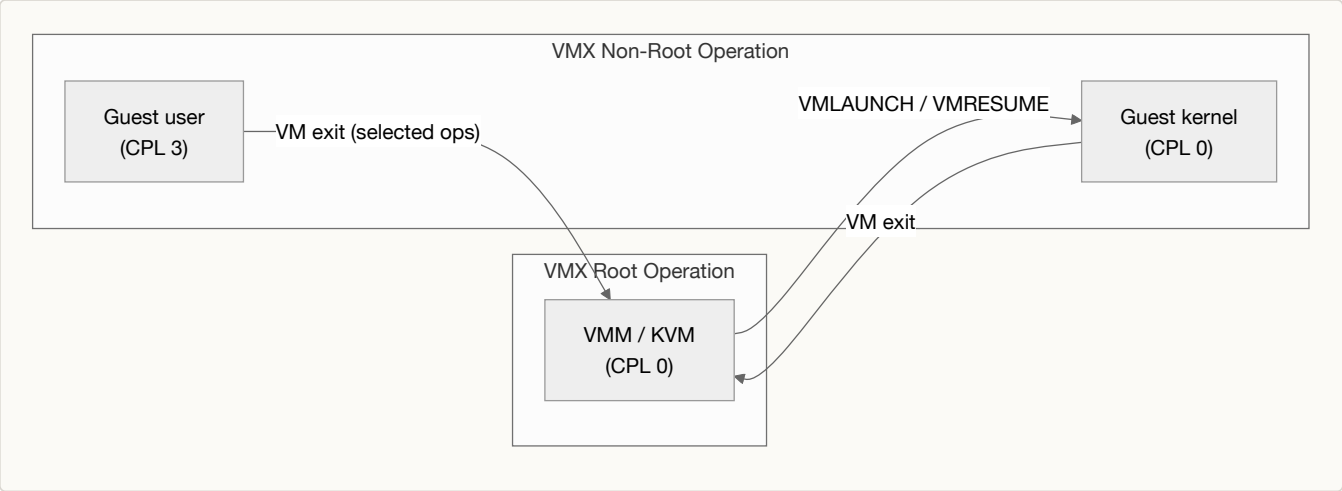
`IA32_VMX_BASIC` carries a few other fields worth naming. Bits 44:32 give the allocation size for VMXON and VMCS regions (1–4096 bytes). Bits 53:50 give the memory type the processor expects for those regions; value 6 means write-back (WB), the value reported by all processors since Nehalem.

On success, `VMXON` transitions the logical processor into **VMX root operation**. The current-VMCS pointer is set to `FFFFFFFF_FFFFFFFFH` (no VMCS active), INIT signals are blocked, and A20M is disabled. The processor will stay in this mode — handling guest entry and exit — until `VMXOFF` returns it to plain IA-32e operation.

VMX Root and Non-Root: A Mode Orthogonal to Rings

The central insight of VT-x is the distinction between **VMX root operation** and **VMX non-root operation**. These two modes exist at a level below the familiar ring hierarchy.

In VMX root operation the VMM executes. The full instruction set is available, including every VMX instruction. In VMX non-root operation the guest executes. Certain operations that would proceed normally in root operation instead cause **VM exits** — a hardware-managed transfer of control back to the VMM. The crucial detail: *there is no software-visible register bit that indicates which mode the CPU is in*. A guest OS running at CPL 0 in VMX non-root operation cannot read a flag and discover it is being virtualized. The mode exists only in the processor's internal state machine, which is exactly what makes it sound as an isolation boundary.



A guest's attempt to execute `VMXOFF` does not switch the processor back to non-VMX mode — it causes a VM exit. The same applies to `VMXON`, `VMLAUNCH`, `VMRESUME`, `VMREAD`, and `VMWRITE`. The VMX instruction set is unconditionally intercepted in non-root operation; no VMCS control bit can allow a guest to use it.

The VMCS

Every virtual CPU (vCPU) in a VT-x system is associated with a **VMCS** — Virtual Machine Control Structure. The VMCS is a processor-managed data structure up to 4096 bytes in size (the exact size is read from `IA32_VMX_BASIC` bits 44:32). Its internal layout is implementation-specific and never documented by Intel; the only portable way to read or write a VMCS field is through `VMREAD` and `VMWRITE`, which take a 32-bit field encoding as their operand.

The first eight bytes have a fixed layout. Bytes 0–3 hold the revision identifier (bits 30:0) and a shadow-VMCS indicator (bit 31): if bit 31 is set, this is a shadow VMCS used for VMCS shadowing, and `VMPTRLD` will reject it unless VMCS shadowing is enabled. Bytes 4–7 are the VMX-abort indicator, written nonzero by the processor if a VMX abort occurs during a VM exit.

Every VMCS has a **launch state** — either "clear" (immediately after `VMCLEAR`) or "launched" (after a successful `VMLAUNCH`). This state is internal to the processor and cannot be read by software; it determines which entry instruction is legal.

VMCS Field Encodings

Every field is addressed by a 32-bit encoding:

Bits	Meaning
`0`	Access type: 0 = full field, 1 = high 32 bits of a 64-bit field
`9:1`	Index within type/width category
`11:10`	Type: 0 = control, 1 = VM-exit info, 2 = guest state, 3 = host state
`14:13`	Width: 0 = 16-bit, 1 = 64-bit, 2 = 32-bit, 3 = natural-width

A few encodings from `arch/x86/include/asm/vmx.h` illustrate the scheme. `PIN_BASED_VM_EXEC_CONTROL` is `0x4000`: type 0 (control), 32-bit, index 0. `EPT_POINTER` is `0x201A`: type 0 (control), 64-bit. `VM_EXIT_REASON` is `0x4402`: type 1 (VM-exit info, read-only), 32-bit. `GUEST_CR0` is `0x6800`: type 2 (guest state), natural-width. `HOST_CR0` is `0x6C00`: type 3 (host state), natural-width.

VMCS Logical Groups

The VMCS groups its fields into six logical areas:

Group	Content
Guest-state area	CR0, CR3, CR4, segment selectors, bases, limits, AR bytes, GDTR, IDTR, RIP, RSP, RFLAGS, DR7, IA32_EFER, IA32_PAT, IA32_DEBUGCTL, activity state, interruptibility state, VMCS link pointer, preemption timer value
Host-state area	CR0, CR3, CR4, segment selectors (CS/SS/DS/ES/FS/GS/TR), FS/GS/TR/GDTR/IDTR bases, RIP, RSP, IA32_EFER, IA32_PAT, IA32_SYSENTER_CS/ESP/EIP
VM-execution control fields	Pin-based controls, primary/secondary processor-based controls, exception bitmap, I/O bitmaps, MSR bitmaps, CR3-target controls, APIC-access address, EPT pointer (`0x201A`), VPID (`0x0000`), preemption timer
VM-exit control fields	VM-exit controls, MSR-store/load areas and counts
VM-entry control fields	VM-entry controls, MSR-load area, event injection field
VM-exit information fields (read-only)	Exit reason (`0x4402`), exit qualification, guest-linear address, guest-physical address, IDT-vectoring info, instruction info, instruction length

The guest-state area is what the processor saves on every VM exit and restores on every VM entry. The host-state area is what the processor loads on every VM exit to hand control back to the VMM. The VMM is responsible for writing host-state fields correctly before the first `VMLAUNCH`; if the processor ever needs to exit and finds garbage in the host-state area, it will jump to a garbage instruction pointer.

VMCS Management Instructions

`VMCLEAR` writes the VMCS to memory, sets its launch state to "clear," and dissociates it from the logical processor. `VMPTRLD` makes a VMCS the current VMCS for the logical processor without changing its launch state. `VMPTRST` stores the current-VMCS pointer to a memory location. `VMREAD` and `VMWRITE` read and write individual fields of the current VMCS by encoding. These instructions are only available in VMX root operation; a guest executing any of them causes a VM exit.

KVM's Use of the VMCS

KVM's VMX backend lives in `arch/x86/kvm/vmx/vmx.c`. The per-vCPU struct is `struct vcpu_vmx`, which embeds `struct kvm_vcpu`. VMCS bookkeeping is tracked by `struct loaded_vmcs` in `arch/x86/kvm/vmx/vmcs.h`. Its `bool launched` field is the flag KVM consults to decide between `VMLAUNCH` and `VMRESUME` for the next guest entry. Its `struct vmcs_host_state host_state` caches

the host CR3, CR4, GS and FS bases, RSP, and segment selectors that KVM would otherwise have to re-read on every entry path — `VMWRITE` is a serializing instruction, so the savings matter at high vCPU counts.

Nested virtualization (KVM hosting a guest that itself runs VMs) introduces three VMCS instances per nested vCPU. `vmcs01` is what KVM builds for the L1 guest hypervisor during normal non-nested operation. `vmcs12` is the VMCS the L1 hypervisor constructs for its L2 nested guest, represented in KVM as `struct vmcs12`. `vmcs02` is the VMCS KVM actually executes L2 with — it merges the policies from `vmcs01` and `vmcs12` so that neither L1 nor KVM can bypass the other's intercept controls.

VM Entry: VMLAUNCH and VMRESUME

`VMLAUNCH` (opcode `0F 01 C2`) performs the first entry into a VMCS. The current VMCS must be in "clear" launch state; on success the processor transitions it to "launched." `VMRESUME` (opcode `0F 01 C3`) performs every subsequent entry and requires the VMCS to already be in "launched" state. Using the wrong instruction — `VMLAUNCH` on a launched VMCS or `VMRESUME` on a clear one — produces a `VMfailValid` with the appropriate error code in the VM-instruction error field.

Both instructions require VMX root operation at CPL 0, `CR0.PE = 1`, and `RFLAGS.VM = 0`. They also require no MOV-SS or POP-SS blocking to be active.

VM entry proceeds through three check phases. Phase 1 validates VMX controls and the host-state area; failure causes `VMfailValid` and leaves guest state unchanged. Phase 2 validates the guest-state area and PDPTRs; failure causes the processor to load the host state and transfer to the host RIP, with bit 31 of the exit-reason field set to indicate a VM-entry failure rather than a true exit. Phase 3 validates the MSR-load area; failure also loads host state. Only after all three phases pass does the processor commit to VMX non-root operation and begin executing guest code.

VM Exits

A VM exit occurs when the guest executes an operation the VMCS execution-control fields have marked for interception, or when the processor encounters a condition that mandates host intervention regardless of control settings — triple fault, NMI, INIT signal, or an external interrupt when `external-interrupt exiting` is set. The processor atomically saves guest state into the VMCS guest-state area, loads host state from the VMCS host-state area, and jumps to the address in the VMCS host RIP field.

The **VM-exit reason field** (`0x4402`, 32-bit, read-only) describes what happened. Bits 15:0 carry the basic exit reason. Bit 31 distinguishes a true VM exit (0) from a VM-entry failure that loaded host state (1). Selected basic reasons:

Code	Reason
0	Exception or NMI
1	External interrupt
2	Triple fault
10	CPUID
12	HLT
18	VMCALL (hypercall)
28	CR access
30	I/O instruction
31	MSR read
32	MSR write
48	EPT violation
49	EPT misconfiguration
52	VMX-preemption timer expired

Exit reasons 48 and 49 deserve a note. They fire when a guest-physical address cannot be resolved through the Extended Page Tables — either because no mapping exists (violation) or because a mapping is present but its permission bits are inconsistent (misconfiguration). Both route to the KVM memory-fault handler, which either populates the mapping or reflects the fault to user space as a `KVM_EXIT_MMIO` exit from `KVM_RUN`. Chapter 5 covers EPT in detail.

Execution Controls and the MSR Bitmap

The most important tool for tuning VM-exit overhead is the **MSR bitmap**. When the primary processor-based VM-execution control "Use MSR bitmaps" (field `0x4002`, bit 28) is set, the processor checks the MSR bitmap before deciding whether an `RDMSR` or `WRMSR` causes an exit. The bitmap is a 4 KiB page: four 1 KiB regions cover MSR reads in `0x00000000–0x00001FFF`, MSR reads in `0xC0000000–0xC0001FFF`, and the corresponding write halves. A set bit means intercept; a clear bit means pass through to the guest. KVM marks the bitmap bits for performance-critical MSRs like `IA32_TSC` (with TSC offsetting active, field `0x4002` bit 3) to avoid exits on every call to `clock_gettime`.

The **VMX-preemption timer** (pin-based control `0x4000` bit 6) provides a deadline mechanism: a 32-bit counter in the VMCS guest-state area decrements proportionally to the TSC (at a rate of one decrement per TSC bit-X transition, where X is read from `IA32_VMX_MISC`). When the counter reaches

zero, a VM exit fires with reason 52. KVM uses this to implement the vCPU preemption timer for guests that spin in HLT loops.

Event Injection

To deliver an interrupt or exception to a guest on the next VM entry, the VMM writes the **VM-entry interruption-information field** in the VMCS. The 32-bit layout: bits 7:0 are the vector; bits 10:8 are the type (0 = external interrupt, 2 = NMI, 3 = hardware exception, 4 = software interrupt, 5 = privileged software exception, 6 = software exception); bit 11 enables error-code delivery; bit 31 marks the field valid. Setting bit 31 to 0 suppresses injection. On the next `VMRESUME`, the processor delivers the event as if it had arrived naturally — through the IDT, with all the privilege checks that entails.

AMD-V (SVM)

Intel shipped the first VT-x–capable processors in November 2005; AMD followed in May 2006 with the Athlon 64 Orleans and Windsor desktop processors, and added Nested Page Tables in the third-generation Opteron "Barcelona" (Family 0x10) in 2007. The architecture is called **SVM** — Secure Virtual Machine — and while it achieves the same isolation goals as VT-x, it makes different tradeoffs that show up clearly in KVM's two backends.

Detecting and Enabling SVM

SVM availability is signaled by `CPUID` leaf `0x80000001`, ECX bit 2 = 1. The feature capability leaf `0x8000000A` gives further detail: EAX returns the SVM revision number, EBX returns the NASID (number of available ASIDs), and EDX carries individual feature bits. The features that matter most in practice are bit 0 (Nested Page Tables), bit 3 (NRIP Save — next sequential RIP recorded in the VMCB on exit, sparing the hypervisor from decoding the faulting instruction), bit 5 (VMCB Clean Bits — allows the CPU to cache VMCB fields across consecutive `VMRUN` calls), and bit 13 (AVIC — Advanced Virtual Interrupt Controller, hardware-accelerated interrupt delivery).

Enabling SVM requires setting `EFER.SVME` (bit 12 of MSR `0xC0000080`). Before doing so, the VMM reads MSR `MSR_VM_CR` at `0xC0010114` and checks bit 4 (`SVMDIS`). If `SVMDIS` is 1, firmware has locked SVM off and it cannot be re-enabled without a power cycle — the lock is asymmetric by design, because some enterprise security policies prohibit guest execution. Finally, the VMM allocates a 4 KiB-aligned **host save area** and writes its physical address to `MSR_VM_HSAVE_PA` at `0xC0010117`. This page is where the CPU will save host state on every `VMRUN`.

The VMCB

AMD's counterpart to the VMCS is the **VMCB** — Virtual Machine Control Block — a 4 KiB page whose physical address the hypervisor passes in `RAX` to the `VMRUN` instruction. Unlike the VMCS, the VMCB is a plain memory-mapped structure with documented field offsets. The hypervisor reads and writes it with

ordinary load and store instructions after mapping it into the host virtual address space. That accessibility makes VMCB manipulation faster than VMCS manipulation (no serializing `VMREAD` / `VMWRITE` pairs) but means KVM must be careful about cache coherency and VMCB clean bits.

The VMCB splits into two halves. The **control area** (bytes `0x000` – `0x3FF`, 1024 bytes) holds everything the CPU consults before guest entry: intercept vectors, TSC offset, IOPM and MSRPM pointers, ASID, TLB controls, interrupt controls, event injection, the NPT root pointer, and VMCB clean bits. The **state save area** (starting at byte `0x400`) holds the full architectural state of the vCPU. The layout, from `arch/x86/include/asm/svm.h`, places segment registers and descriptors at `0x400`, EFER at `0x4D0`, CR4 at `0x548`, CR3 at `0x550`, CR0 at `0x558`, RFLAGS at `0x570`, RIP at `0x578`, RSP at `0x5D8`, RAX at `0x5F8`, the SYSCALL MSRs (STAR= `0x600`, LSTAR= `0x608`, CSTAR= `0x610`, SFMASK= `0x618`, KERNEL_GS_BASE= `0x620`), g_PAT at `0x668`, and SPEC_CTRL at `0x6E0`. The struct is 744 bytes in total. CET state fields (s_cet, ssp, isst_addr) sit between RSP and RAX, which is why the latter fields are offset significantly further than they appear in earlier versions of the header.

Intercept Controls

At VMCB offset `0x000`, six consecutive 32-bit words (192 bits total) form the intercept bitmap. Each bit controls whether a specific guest operation triggers `#VMEXIT`. KVM manipulates these through `vmcb_set_intercept()` and `vmcb_clr_intercept()` from `arch/x86/include/asm/svm.h`.

Selected flat bit indices: 96 (external interrupt), 97 (NMI), 107 (CPUID), 120 (HLT), 123 (IOIO — I/O port access), 124 (MSR_PROT — MSR access controlled by the MSRPM), 127 (SHUTDOWN — triple fault), 128 (VMRUN — always intercepted in any nested-SVM setup so a guest hypervisor cannot execute `VMRUN` directly), 129 (`VMMCALL` — hypercall), and 141 (XSETBV).

The I/O and MSR permission maps work analogously to the VT-x bitmaps. The **MSRPM** is 8 KiB at 4 KiB alignment: four 2 KiB regions covering MSR ranges `0x00000000–0x1FFF`, `0xC0000000–0xC0001FFF`, and `0xC0010000–0xC0011FFF`, with 2 bits per MSR (read intercept and write intercept). The **IOPM** is 12 KiB with one bit per I/O port. KVM programs both during vCPU creation and updates them as the virtual device model grows.

VMCB Clean Bits

On CPUs where `CPUID 0x8000000A` EDX bit 5 is set, the processor can cache VMCB field groups across consecutive `VMRUN` calls. The **clean bits** field at VMCB offset `0x0C0` acts as a validity bitmap: when a bit is set, the CPU is permitted to use its cached copy instead of re-reading from memory. The hypervisor must clear any bit whose corresponding fields it has modified since the last `VMRUN`.

Selected bits from `arch/x86/kvm/svm/svm.h`:

Bit	Covers
`VMCB_INTERCEPTS` (0)	Intercept vectors, TSC offset, pause filter
`VMCB_PERM_MAP` (1)	IOPM and MSRPM base addresses
`VMCB_ASID` (2)	ASID
`VMCB_INTR` (3)	Interrupt control fields
`VMCB_NPT` (4)	NPT enable, `nested_cr3`, `g_PAT`
`VMCB_CR` (5)	CR0, CR3, CR4, EFER
`VMCB_SEG` (8)	CS, DS, SS, ES, CPL

KVM clears the appropriate bits whenever it modifies a field group, and sets them all at the end of a successful `#VMEXIT` handler so the next `VMRUN` can take maximum advantage of caching. On a busy system where most exits are IOIO or MSR faults with no CR changes, bits 0, 1, 3, 4, 5, and 8 can survive across hundreds of consecutive entries, materially reducing the cost of each `VMRUN`.

VMRUN and #VMEXIT

`VMRUN rAX` (opcode `0F 01 D8`) is the SVM instruction that corresponds to both `VMLAUNCH` and `VMRESUME` combined. There is no separate "first entry" instruction: the CPU loads guest state from the VMCB, applies control fields, and begins executing guest code. Host state is saved to the `MSR_VM_HSAVE_PA` page automatically — the fields saved are SS selector, RSP, CR0, CR3, CR4, EFER, IDTR, and GDTR.

On `#VMEXIT`, the processor writes guest state back into the VMCB state save area, records the exit reason at VMCB offset `0x070` (`exit_code`), stores additional qualification at `0x078` (`exit_info_1`) and `0x080` (`exit_info_2`), restores host state from `MSR_VM_HSAVE_PA`, and jumps to the host `#VMEXIT` handler.

Not everything goes through `VMRUN`'s automatic save mechanism. `VMSAVE rAX` (opcode `0F 01 DB`) and `VMLOAD rAX` (opcode `0F 01 DA`) handle extended state: FS/GS base, LDTR, TR, STAR, LSTAR, CSTAR, SFMASK, KernelGsBase, and the SYSENTER MSRs. KVM calls `VMSAVE` before `VMRUN` to capture any host extended state, and `VMLOAD` after to restore it. Forgetting this step would leave the host's FS base mapped as the guest's FS base after the first `VMRUN`, a privilege crossing that no software check would catch.

If the NRIP feature is present (`CPUID 0x8000000A` EDX bit 3), the processor records the next sequential instruction address at VMCB offset `0x0C8` (`next_rip`) on every `#VMEXIT`. KVM uses this to skip the faulting instruction when emulating I/O port accesses and similar traps, avoiding a full instruction decode.

Selected `#VMEXIT` exit codes from `arch/x86/include/uapi/asm/svm.h`:

Constant	Value	Meaning
`SVM_EXIT_EXCP_BASE`	`0x040`	Exception base (vectors 0–31 at `0x040` – `0x05F`)
`SVM_EXIT_INTR`	`0x060`	External interrupt
`SVM_EXIT_NMI`	`0x061`	NMI
`SVM_EXIT_VINTR`	`0x064`	Virtual interrupt window open
`SVM_EXIT_CPUID`	`0x072`	CPUID
`SVM_EXIT_HLT`	`0x078`	HLT
`SVM_EXIT_IOIO`	`0x07B`	I/O port access
`SVM_EXIT_MSR`	`0x07C`	MSR access
`SVM_EXIT_SHUTDOWN`	`0x07F`	Triple fault
`SVM_EXIT_VMRUN`	`0x080`	Guest executed VMRUN
`SVM_EXIT_VMMCALL`	`0x081`	Hypercall
`SVM_EXIT_NPF`	`0x400`	Nested page fault
`SVM_EXIT_VMGEXIT`	`0x403`	SEV-ES VMGEXIT

`SVM_EXIT_NPF` at `0x400` is AMD's nested-page-fault exit — the equivalent of Intel's EPT violation (exit reason 48). `exit_info_1` carries the fault-error bits and `exit_info_2` carries the guest-physical address that faulted.

ASIDs and TLB Management

SVM uses **ASIDs** (Address Space Identifiers) to tag TLB entries per guest, preventing cross-VM TLB pollution and avoiding full flushes on every `VMRUN`. ASID 0 is reserved for the host. The maximum ASID is `CPUID(0x8000000A).EBX - 1` (NASID minus one). The ASID is set in the VMCB control area at offset `0x058` and the TLB flush mode is set at `0x05C` (`tlb_ctl`):

Value	Meaning
0	`TLB_CONTROL_DO_NOTHING` — reuse existing TLB entries
1	`TLB_CONTROL_FLUSH_ALL_ASID` — flush all TLB entries with this ASID
3	`TLB_CONTROL_FLUSH_ASID` — flush non-global entries for this ASID
7	`TLB_CONTROL_FLUSH_ASID_LOCAL` — flush on this logical CPU only

When the per-CPU ASID counter exhausts the pool (`next_asid > max_asid`), KVM increments its generation counter and writes `TLB_CONTROL_FLUSH_ALL_ASID` to force a clean slate on the next `VMRUN`. Intel's analogue is the VPID (16-bit, VMCS field `0x0000`) and the `INVVPID` instruction.

Event Injection

Event injection on AMD uses `event_inj` (VMCB offset `0x0A8`, 32-bit). The layout is structurally identical to VT-x: bits 7:0 are the vector, bits 10:8 are the type (0 = hardware interrupt, 2 = NMI, 3 = exception, 4 = software interrupt), bit 11 is the error-code-valid flag, bit 31 marks the field valid. When bit 31 is set on entry into `VMRUN`, the processor delivers the event to the guest before executing its first instruction.

The interrupt-window mechanism differs between the two architectures. VT-x uses a dedicated primary processor-based control (field `0x4002`, bit 2, "interrupt-window exiting") that causes an immediate VM exit when the guest reaches an interruptible state (RFLAGS.IF = 1, no blocking). AMD uses the `V_IRQ` bit in the `int_ctl` field (VMCB offset `0x060`), which signals that a virtual interrupt is pending; when the guest becomes interruptible, the processor fires `SVM_EXIT_VINTR` (`0x064`). The end result is the same — the VMM gets a callback at the first moment it is safe to inject — but the mechanism differs.

Nested Page Tables

AMD's second-level address translation is called **NPT** (Nested Page Tables), also marketed as RVI (Rapid Virtualization Indexing). It was introduced with the "Barcelona" Family 0x10 Opteron. Enabling NPT requires setting the `SVM_MISC_ENABLE_NP` bit in the VMCB `misc_ctl` field at offset `0x090`, and writing the host-physical address of the nested page-table root into `nested_cr3` at offset `0x0B0`. KVM additionally clears the `INTERCEPT_INVLPG` bit and removes `PF_VECTOR` from the exception intercept bitmap when NPT is active — with NPT, page faults inside the guest no longer need to exit, because the hardware resolves guest-physical to host-physical without software involvement.

One documented asymmetry between AMD NPT and Intel EPT: AMD NPT does not support execute-only mappings. An NPT entry with execute permission set must also have read permission. Intel EPT permits execute-only pages (XWR bits `0b100`). This asymmetry surfaces in KVM's NPT entry construction and in any hypervisor that tries to use execute-only guard pages for shadow-stack hardening.

VT-x and SVM Side by Side

The two architectures solve the same problem with the same primitives but make opposite tradeoffs on structure access:

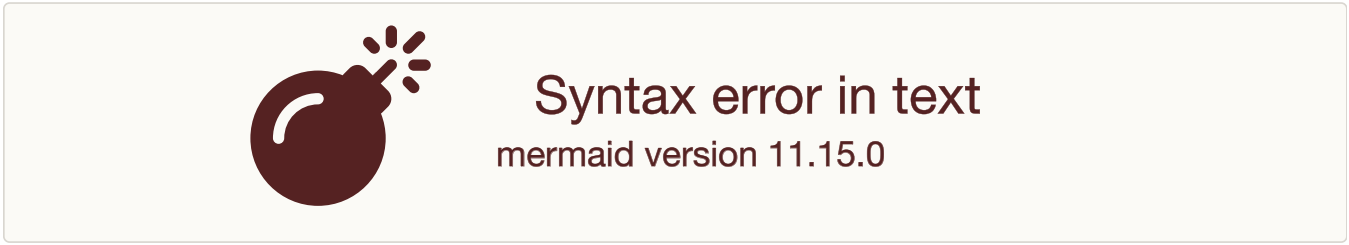
Aspect	Intel VT-x	AMD SVM
Control block	VMCS (opaque, `VMREAD` / `VMWRITE`)	VMCB (4 KiB memory-mapped struct)
Entry instruction	`VMLAUNCH` / `VMRESUME`	`VMRUN rAX`
Exit reason location	VMCS `VM_EXIT_REASON` (`0x4402`)	VMCB `exit_code` offset `0x070`
Host state save	VMCS host-state area	`MSR_VM_HSAVE_PA` physical page
TLB tagging	VPID (16-bit, VMCS `0x0000`)	ASID (32-bit, VMCB `0x058`)
SLAT	EPT (execute-only pages supported)	NPT (execute-only not supported)
Interrupt window	Primary control bit 2	`V_IRQ` in `int_ctl` → `SVM_EXIT_VINTR`
Hypercall	`VMCALL`	`VMMCALL`
Next-RIP on exit	VM-exit instruction-length VMCS field	`next_rip` at VMCB `0x0C8` (NRIP feature)

The structural difference matters to KVM's two backends (`arch/x86/kvm/vmx/vmx.c` and `arch/x86/kvm/svm/svm.c`), which share a common `struct kvm_vcpu` core but diverge entirely on how they program and read control state. From the guest's perspective — the one running at CPL 0 in non-root operation — the difference is invisible.

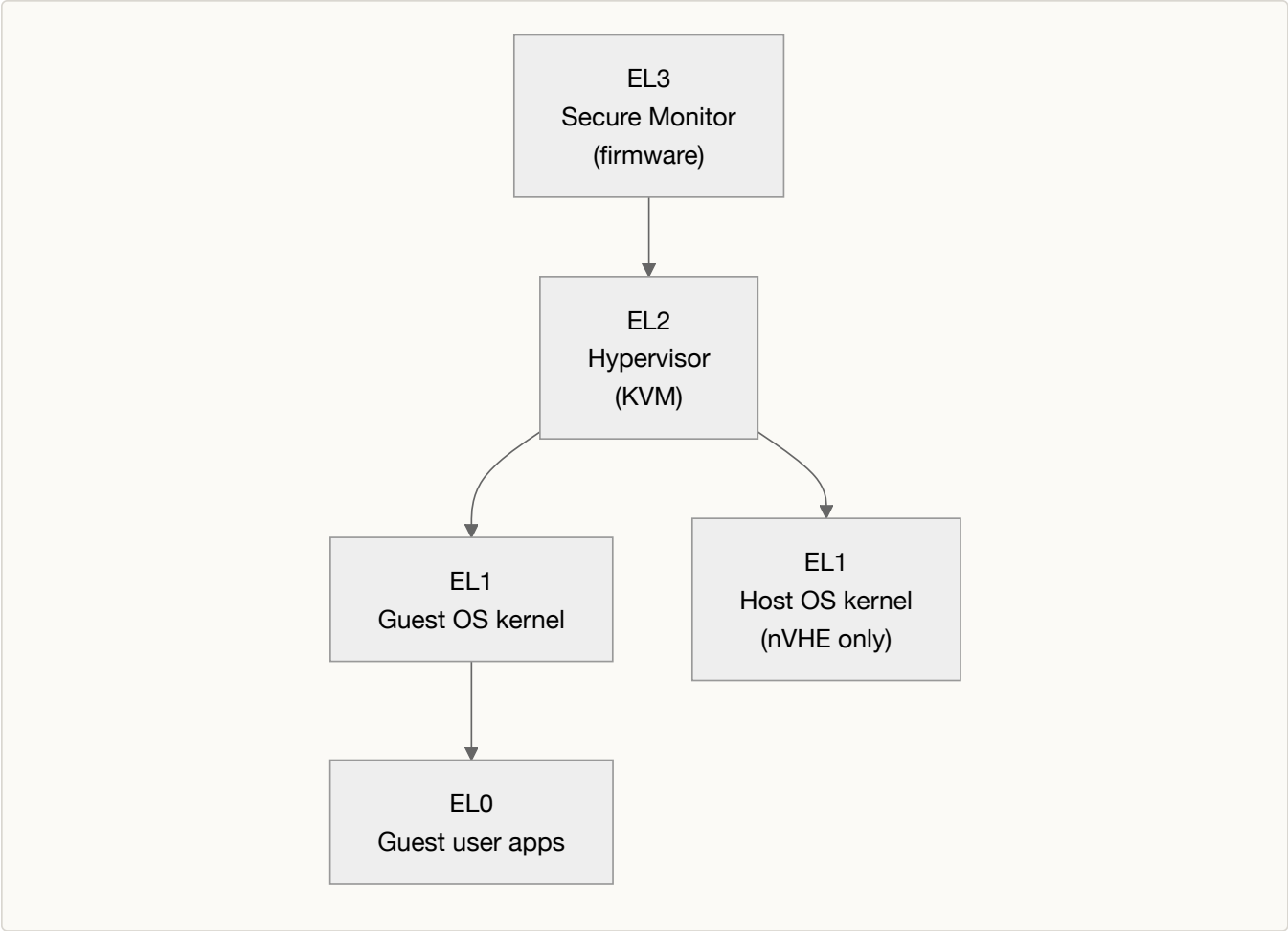
ARM Virtualization Extensions

Exception Levels

AArch64 organizes privilege into four **exception levels**:



EL0 and EL1 are mandatory on every AArch64 implementation. EL2 and EL3 are optional; hardware that omits EL2 provides no virtualization extensions and cannot run KVM. Code at EL0 cannot access system registers at all. Code at EL1 can access EL1 system registers but not EL2 or EL3 registers. EL2 can access EL1 registers (to save and restore guest context) and has its own set of hypervisor registers — `HCR_EL2`, `VTCR_EL2`, `VTTBR_EL2`, and others — that EL1 cannot read or write.



This is not the x86 root/non-root distinction. There is no separate mode bit layered below the ring hierarchy. The EL hierarchy *is* the isolation boundary. A guest OS at EL1 simply does not have the instruction encodings to write EL2 registers. Any attempt raises an exception that routes to EL2 rather than executing.

Exceptions in AArch64 can only move to the same level or a higher level on entry. `ERET` can only return to the same or a lower level. A guest kernel at EL1 issuing an `ERET` cannot route execution to EL2 — it would return to EL0. The hardware exception model makes EL2 a strict parent of EL1, not a peer reachable from below.

HCR_EL2

`HCR_EL2`, the **Hypervisor Configuration Register**, is a 64-bit system register at EL2 that governs what EL1 and EL0 can do. It is the primary control surface for a KVM vCPU, roughly analogous to the VMCS execution-control fields for VT-x.

The most important bit for basic virtualization is bit 0, **VM**: when set, the CPU enables stage-2 address translation for the EL1&0 regime, translating Intermediate Physical Addresses (what the guest calls physical memory) to Host Physical Addresses (the real DRAM locations). Clearing bit 0 makes the guest's

physical addresses resolve directly to host-physical — appropriate only during early bring-up when the hypervisor is not yet protecting guest memory.

Bits 3 (**FMO**), 4 (**IMO**), and 5 (**AMO**) route physical FIQ, IRQ, and SError exceptions to EL2, preventing the guest from seeing raw hardware interrupts and allowing the hypervisor to inject virtual IRQs through the GIC instead.

Bit 13 (**TWI**) traps `WFI` (Wait For Interrupt) from EL0 and EL1 to EL2. This is the ARM equivalent of VT-x's HLT exiting (exit reason 12). When a guest vCPU executes `WFI` to idle, KVM gets control and can schedule another vCPU or yield the physical core.

Bit 18 (**TID3**) traps EL1 reads of the group-3 ID registers — the registers advertising CPU features, implementation options, and ISA revisions — to EL2. KVM intercepts these reads and returns synthesized values, which is the mechanism behind Firecracker's **V1N1 static CPU template** on ARM: a host running on an AWS Graviton (Neoverse V1 microarchitecture) presents itself to the guest as Neoverse N1, improving migration portability across instance types. Firecracker's V1N1 template requires host KVM capabilities `KVM_CAP_ARM_PTRAUTH_ADDRESS` (171) and `KVM_CAP_ARM_PTRAUTH_GENERIC` (172) so it can safely expose or suppress pointer-authentication features.

Bit 31 (**RW**) sets the execution state for EL1: 1 means AArch64, 0 means AArch32. Every 64-bit Linux guest needs this set to 1.

Bit 34 (**E2H**) enables Virtualization Host Extensions. Bit 46 (**FWB**, added in ARMv8.4) allows stage-2 attributes to directly override stage-1 cacheability, giving the hypervisor direct control over guest memory type without the guest being able to influence it.

Stage-2 Address Translation

When `HCR_EL2.VM = 1`, every memory access from EL1 or EL0 goes through two independent MMU walks. The guest OS programs its own page tables as always, translating guest-virtual addresses to what it believes are physical addresses — ARM calls these **Intermediate Physical Addresses (IPA)**. The hardware then performs a second walk, controlled by the hypervisor, translating each IPA to a **Host Physical Address (HPA)**.

`VTCR_EL2` (Virtualization Translation Control Register) configures the stage-2 walk: `T0SZ` (bits 5:0) defines the IPA input range as `2^(64–T0SZ)` bytes; `SL0` (bits 7:6) sets the starting lookup level; `TG0` (bits 15:14) selects the translation granule (4 KB = `0b00`, 64 KB = `0b01`, 16 KB = `0b10`); `PS` (bits 18:16) sets the output address size.

`VTTBR_EL2` carries two things: the VMID in bits [63:48] (for 16-bit VMIDs, ARMv8.1+) or [55:48] (for 8-bit VMIDs, ARMv8.0), and the stage-2 page-table base address in bits [47:1]. The VMID tags TLB entries per VM exactly as ASID does for AMD or VPID does for Intel: world-switching between two VMs does not require flushing the TLB so long as each VM has a distinct VMID.

The stage-2 walk is independent of stage-1. The hypervisor can disable or re-enable stage-2 independently, which is useful during early boot when the hypervisor is initializing a guest's memory map before enabling the full translation regime.

VHE: Running the Host Kernel at EL2

Without Virtualization Host Extensions, the standard AArch64 arrangement places the KVM hypervisor stub at EL2 and the host Linux kernel at EL1. Every guest entry and exit requires a full world switch: save all EL1 host registers, load all EL1 guest registers, `ERET` to guest EL1 on entry; reverse the process on exit, returning to EL2, then dropping back to EL1 host context. The host and guest EL1 contexts are completely symmetric — both are "just an EL1" — but they cannot coexist on the CPU simultaneously.

VHE (Virtualization Host Extensions), introduced in ARMv8.1-A, collapses this asymmetry. When `HCR_EL2.E2H = 1` (bit 34), the CPU enters a mode where EL2 becomes a superset of EL1: the host kernel can run directly at EL2 with full OS semantics. Most EL1 system registers accessed from EL2 redirect to their EL2 equivalents — `SCTLR_EL1` at EL2 accesses `SCTLR_EL2`, and so on. New `_EL12` aliases (`SCTLR_EL12`, `TCR_EL12`, `TTBR0_EL12`, `VBAR_EL12`) give the hypervisor access to the actual EL1 register contents for guest context save/restore, without confusion from the E2H redirect.

A companion bit, `HCR_EL2.TGE` (bit 27), switches user-space semantics. When `TGE = 1` alongside `E2H = 1`, all physical exceptions from EL0 route to EL2 — the EL0 threads are treated as host-OS user processes. When `TGE = 0`, EL0 threads are guest user space, with exceptions routing to EL1 as normal.

Linux KVM on ARMv8.1+ runs the host kernel at EL2 using VHE. The same kernel binary supports both VHE and non-VHE (nVHE) via runtime alternative instruction patching decided at boot based on CPU feature detection. Starting with ARMv9.5, implementations may make `HCR_EL2.E2H` a RES1 field — permanently 1, making VHE the only implemented behavior and removing the non-VHE code path from relevance on new silicon.

```
flowchart LR
  subgraph nvhe["nVHE (ARMv8.0)"]
    el2s["KVM stub at EL2"]
    el1h2["Host kernel at EL1"]
    el1g2["Guest at EL1"]
    el2s -->|"world switch"| el1g2
    el2s --> el1h2
  end
  subgraph vhe["VHE (ARMv8.1+, E2H=1)"]
    el2v["Host kernel + KVM at EL2"]
    el1gv["Guest at EL1"]
    el2v -->|"ERET / exception"| el1gv
  end
```

Exit Handling on ARM

When the guest triggers an exception that routes to EL2, the syndrome register `ESR_EL2` records what happened. Bits 31:26 (`EC` field) carry the exception class. KVM's `arm_exit_handlers[]` array in `arch/arm64/kvm/handle_exit.c` maps EC values to handler functions via `kvm_get_exit_handler()`.

Selected EC codes:

Symbol	Cause
`ESR_ELx_EC_WFx`	WFI/WFE trap — guest idle
`ESR_ELx_EC_HVC64`	HVC — hypervisor call from guest
`ESR_ELx_EC_SMC64`	SMC — secure monitor call
`ESR_ELx_EC_SYS64`	System register access trap
`ESR_ELx_EC_DABT_LOW`	Data abort / MMIO / stage-2 fault
`ESR_ELx_EC_IABT_LOW`	Instruction abort

`ESR_ELx_EC_DABT_LOW` is the ARM equivalent of an EPT violation: a stage-2 fault on a data access. The handler walks the ESR fields to determine whether the fault is MMIO (no mapping exists and the address is in a device region) or a genuine page fault (a mapping needs to be installed), then either emulates the device access or populates the stage-2 table.

GIC Virtualization

ARM guests need to receive interrupts. The Generic Interrupt Controller (GIC) has had virtualization support since GICv2, adding a two-register-bank split: the **virtual interface control block** (`GICH_*` registers) that the hypervisor programs, and the **virtual CPU interface** (`GICV_*` registers) that the guest reads as if they were the physical GIC CPU interface registers.

Virtual interrupt delivery uses `GICH_LRn` — the List Registers. Each LR entry specifies a virtual interrupt ID, an optional physical interrupt ID (when `HW = 1` for hardware-backed IRQs), a priority, a state (pending, active, or pending+active), and EOI behavior. The number of list registers is reported by `GICH_VTR`, typically between 4 and 16 on real hardware. `GICH_HCR` bit 0 (`En`) must be set before guest entry or no virtual interrupts will be delivered.

When `HW = 1` in a list register, hardware deactivates the physical interrupt automatically when the guest writes EOI to `GICV_EOIR`. When `HW = 0`, the hypervisor must manually deactivate the physical IRQ — usually by writing the physical INTID to `GICD_DIR` — after the guest's interrupt handler runs.

GICv3 and GICv4 move the interface to system registers: `ICH_HCR_EL2` replaces `GICH_HCR`, `ICH_LR<n>_EL2` (up to 16) replace `GICH_LRn`, and the virtual CPU interface becomes `ICV_*` registers at EL1. The architecture is part of the AArch64 architectural state from ARMv8 onward; GICv3 support is

mandatory for any arm64 server platform.

Firecracker on aarch64 requires `KVM_CAP_IRQCHIP` — in-kernel GICv2 or GICv3 emulation — in addition to `/dev/kvm`. Hardware with `CONFIG_KVM=y` but no IRQ chip capability at runtime cannot host Firecracker guests; this was confirmed by Firecracker issue #1186, where a Raspberry Pi 3 built with KVM enabled failed the capability check at startup.

PSCI

Guest power management on ARM does not use ACPI PM registers. Instead, the guest issues **PSCI** (Power State Coordination Interface, ARM DEN0022) calls using the SMCCC (DEN0028) calling convention. The conduit is SMC when EL3 is present, or HVC when only EL2 is available.

KVM emulates PSCI via `kvm_smccc_call_handler()`. Selected 64-bit function IDs: `CPU_ON` is `0xC4000003`, `CPU_SUSPEND` is `0xC4000001`, `SYSTEM_OFF` is `0x84000008`, and `SYSTEM_RESET` is `0x84000009`. KVM exposes PSCI to guests via the `KVM_ARM_VCPU_PSCI_0_2` feature flag set on the vCPU. For events it cannot handle internally — `SYSTEM_OFF` and `SYSTEM_RESET` — KVM returns `KVM_EXIT_SYSTEM_EVENT` to user space. In Firecracker, a `poweroff` command inside the guest triggers a PSCI `SYSTEM_OFF` call, KVM emits `KVM_EXIT_SYSTEM_EVENT`, and Firecracker performs a clean shutdown, unmounting storage and releasing resources before the VMM process exits.

How Silicon Enforces the Boundary

The same question runs through all three architectures: what actually prevents guest ring 0 from becoming host ring 0? The answer is not a software invariant or an operating system convention — it is hardware decode logic that the guest cannot observe or bypass.

On x86 with VT-x, the CPU maintains an internal mode bit distinguishing VMX root from VMX non-root operation. Guest code executing at CPL 0 in VMX non-root cannot use VMX instructions — they cause VM exits, not execution. It cannot read or write the VMCS — `VMREAD` and `VMWRITE` are unconditionally intercepted. It cannot execute `VMXOFF` or `VMXON` — also unconditionally intercepted. The guest's view of physical memory is confined to what EPT maps, and EPT is controlled entirely by the VMM running in root mode. The guest kernel therefore operates at full CPL 0 privilege within its architectural world, but that world is bounded by hardware: nothing the guest can do in non-root mode reaches host state or host memory.

On AMD-V, the `VMRUN` intercept bit (index 128 in the VMCB intercept bitmap) is required in any system where a guest hypervisor might try to execute `VMRUN` itself. Every `VMRUN` the guest attempts fires a `#VMEXIT` back to the real hypervisor. The VMCB resides in host-physical memory that NPT does not expose to the guest once the nested page tables are active. Host state is saved to and restored from the `MSR_VM_HSAVE_PA` page, which is likewise outside the guest's NPT-visible address space.

On ARM, the mechanism is the exception level hierarchy itself. A guest OS at EL1 does not have instruction encodings that write EL2 registers. Writing `HCR_EL2` requires EL2; the guest cannot disable stage-2 translation by clearing `HCR_EL2.VM`. Writing `VTTBR_EL2` to install new stage-2 page tables requires EL2; the guest cannot remap the hypervisor's memory into its own address space. Any EL1 instruction that attempts to access an EL2-only register raises an exception that routes to EL2 — to the hypervisor — not back to EL1. The hardware exception model guarantees that exceptions from one VM's EL1 cannot route to another VM's EL1; they always traverse EL2 first.

Chapter 5 builds on this foundation — EPT and NPT give the hypervisor the same kind of structural control over the guest's view of physical memory that the mode bit and exception-level hierarchy provide over instruction execution.

Sources And Further Reading

- **Intel SDM Vol 3C** (virtualization chapter, primary reference for VT-x, VMCS structure, VMLAUNCH/VMRESUME, VM-exit reason table): https://cdrdv2-public.intel.com/789585/326019-sdm-vol-3c.pdf
- **Intel SDM Vol 3 combined**: https://cdrdv2-public.intel.com/671447/325384-sdm-vol-3abcd.pdf
- **felixcloutier x86 instruction reference — VMLAUNCH/VMRESUME**: https://www.felixcloutier.com/x86/vmlaunch:vmresume
- **felixcloutier — VMXON**: https://www.felixcloutier.com/x86/vmxon
- **felixcloutier — VMREAD**: https://www.felixcloutier.com/x86/vmread
- **felixcloutier — CPUID**: https://www.felixcloutier.com/x86/cpuid
- **Intel SDM rendered — VMCS region format**: https://xem.github.io/minix86/manual/intel-x86-and-64-manual-vol3/o_fe12b1e2a880e0ce-1049.html
- **Intel SDM rendered — VMCS field encoding table**: https://xem.github.io/minix86/manual/intel-x86-and-64-manual-vol3/o_fe12b1e2a880e0ce-1071.html
- **Intel SDM rendered — VM-exit reason field**: https://xem.github.io/minix86/manual/intel-x86-and-64-manual-vol3/o_fe12b1e2a880e0ce-1067.html
- **Intel SDM rendered — exit reason table (Appendix C)**: https://xem.github.io/minix86/manual/intel-x86-and-64-manual-vol3/o_fe12b1e2a880e0ce-1961.html
- **Intel SDM rendered — IA32_VMX_BASIC**: https://xem.github.io/minix86/manual/intel-x86-and-64-manual-vol3/o_fe12b1e2a880e0ce-1943.html
- **Linux kernel** `arch/x86/include/asm/vmx.h` (VMCS field encodings): https://github.com/torvalds/linux/blob/master/arch/x86/include/asm/vmx.h
- **Linux kernel** `arch/x86/kvm/vmx/vmcs.h` (`struct loaded_vmcs`, `vmcs01/12/02`): https://github.com/torvalds/linux/blob/master/arch/x86/kvm/vmx/vmcs.h

- **Linux kernel `arch/x86/kvm/vmx/vmx.c`** (KVM VT-x backend): https://github.com/torvalds/linux/blob/master/arch/x86/kvm/vmx/vmx.c
- **KVM nested VMX documentation**: https://docs.kernel.org/virt/kvm/x86/nested-vmx.html
- **axvisor VMX definitions and exit-reason types**: https://arceos-hypervisor.github.io/axvisor-crates-book/x86_vcpu/VMX_Definitions_and_Types.html
- **Linux kernel `arch/x86/include/asm/svm.h`** (VMCB struct, intercept enums, clean bits, EVENTINJ): https://github.com/torvalds/linux/blob/master/arch/x86/include/asm/svm.h
- **Linux kernel `arch/x86/include/uapi/asm/svm.h`** (`SVM_EXIT_*` codes): https://github.com/torvalds/linux/blob/master/arch/x86/include/uapi/asm/svm.h
- **Linux kernel `arch/x86/include/asm/msr-index.h`** (MSR addresses including `MSR_VM_CR`, `MSR_VM_HSAVE_PA`, `EFER`): https://github.com/torvalds/linux/blob/master/arch/x86/include/asm/msr-index.h
- **Linux kernel `arch/x86/kvm/svm/svm.c`** (ASID allocation, NPT init, `VMRUN` / `VMSAVE` / `VMLOAD` path): https://github.com/torvalds/linux/blob/master/arch/x86/kvm/svm/svm.c
- **Linux kernel `arch/x86/kvm/svm/svm.h`** (VMCB clean bits enum): https://github.com/torvalds/linux/blob/master/arch/x86/kvm/svm/svm.h
- **raw-cpuid crate `SvmFeatures`** (CPUID `0x8000000A` EDX bits): https://docs.rs/raw-cpuid/latest/raw_cpuid/struct.SvmFeatures.html
- **AMD APM Vol 2** (doc 24593, authoritative VMCB reference): https://www.amd.com/system/files/TechDocs/24593.pdf
- **NoirVisor SVM core — NPT execute-only limitation**: https://github.com/Zero-Tang/NoirVisor/blob/master/src/svm_core/readme.md
- **ARM AArch64 Exception Model v1.3**: https://documentation-service.arm.com/static/63a065c41d698c4dc521cb1c
- **ARM Developer — Exception levels**: https://developer.arm.com/documentation/dui0801/l/Overview-of-AArch64-state/Exception-levels
- **ARM HCR_EL2 register reference (DDI0601)**: https://developer.arm.com/documentation/ddi0601/latest/AArch64-Registers/HCR-EL2--Hypervisor-Configuration-Register
- **Jon Palmisc ARM register reference — HCR_EL2**: https://arm.jonpalmisc.com/latest_sysreg/AArch64-hcr_el2
- **ARM Learn the Architecture — Stage-2 translation**: https://developer.arm.com/documentation/102142/0100/Stage-2-translation
- **ARM Learn the Architecture — VHE**: https://developer.arm.com/documentation/102142/0100/Virtualization-host-extensions
- **LWN — arm64 VHE patch series**: https://lwn.net/Articles/650524/
- **LWN — FEAT_E2H0, ARMv9.5 E2H=RES1**: https://lwn.net/Articles/959153/

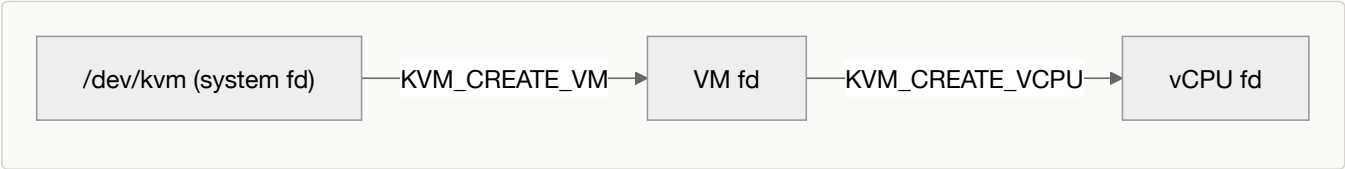
- **Linux kernel** `arch/arm64/kvm/handle_exit.c` (EC code handlers, `arm_exit_handlers[]`): https://github.com/torvalds/linux/blob/master/arch/arm64/kvm/handle_exit.c
- **KVM ARM PSCI documentation**: https://www.kernel.org/doc/html/v5.6/virt/kvm/arm/psci.html
- **arm-trusted-firmware** `psci.h` (PSCI function IDs): https://github.com/96boards-poplar/arm-trusted-firmware/blob/master/include/lib/psci/psci.h
- **ARM GICv2 List Registers IHI0048B**: https://developer.arm.com/documentation/ihi0048/b/GIC-Support-for-Virtualization/GIC-virtual-interface-control-registers/List-Registers--GICH-LRn
- **OSDev — GICv3/v4**: https://wiki.osdev.org/Generic_Interrupt_Controller_versions_3_and_4
- **Firecracker FAQ**: https://github.com/firecracker-microvm/firecracker/blob/main/FAQ.md
- **Firecracker CPU templates documentation** (V1N1 ARM template, TID3 mechanism): https://github.com/firecracker-microvm/firecracker/blob/main/docs/cpu_templates/cpu-templates.md
- **Firecracker issue #1186** (aarch64 KVM IRQ chip capability requirement): https://github.com/firecracker-microvm/firecracker/issues/1186

Chapter 5: The KVM API

You have a CPU that supports VMX non-root mode and a kernel module, `kvm.ko`, that can put it there. The question now is how userspace talks to that module. The kernel could expose KVM through a sysfs hierarchy, a netlink socket, a purpose-built syscall, or a device file. It chose the last: a single character device at `/dev/kvm` whose entire surface is made of `ioctl` calls. That choice has consequences for how a VMM is structured — and for how you read one. The reference VMM throughout is Firecracker; the canonical teaching program is the LWN article by Josh Triplett, published September 29, 2015.

Three File Descriptor Scopes

The KVM ioctl surface divides cleanly into three scopes, each represented by a different file descriptor. Mixing them is not silently wrong — the kernel returns `ENOTTY`, which is the right error for "this ioctl does not apply to this file descriptor." The structure forces the VMM to build its object graph in a fixed order: a system fd produces a VM fd, which produces vCPU fds. None of the objects can be created out of sequence.



Every KVM ioctl number is constructed using the `_IO`, `_IOR`, or `_IOW` macros with the magic byte `KVMIO = 0xAE`. That byte is the namespace; the low byte is the command. `KVM_GET_API_VERSION` is `_IO(0xAE, 0x00)`, `KVM_CREATE_VM` is `_IO(0xAE, 0x01)`, `KVM_RUN` is `_IO(0xAE, 0x80)`. The full reference table appears at the end of this chapter.

The system fd covers VM lifecycle and capability queries. The VM fd covers memory configuration, interrupt-controller setup, and event fd wiring. The vCPU fd covers execution and register access. Two threading rules matter: VM ioctls must be issued from the same process address space used to create the VM, and vCPU ioctls should be issued from the same OS thread that created the vCPU — Firecracker enforces the latter by running each vCPU in its own dedicated OS thread named `fc_vcpu {index}`.

The API Version Handshake

Before issuing any other ioctl, a VMM opens `/dev/kvm` with `O_RDWR | O_CLOEXEC` and checks the API version:

```
int kvmfd = open("/dev/kvm", O_RDWR | O_CLOEXEC);
int api_version = ioctl(kvmfd, KVM_GET_API_VERSION, NULL);
```

`KVM_GET_API_VERSION` returns the integer `12`. It has returned `12` since the API was stabilized in Linux 2.6.22, and the kernel documentation states it is not expected to change. The KVM documentation says applications must refuse to run if they receive any other value. Firecracker checks this immediately at startup — `kvm_fd.get_api_version() != 12` triggers an early abort — because the rest of the initialization sequence assumes version 12 semantics throughout.

Once version 12 is confirmed, all ioctls documented with the capability tag "basic" are guaranteed available without further negotiation. Capabilities beyond "basic" require explicit `KVM_CHECK_EXTENSION` queries.

Capability Negotiation

`KVM_CHECK_EXTENSION` is `_IO(0xAE, 0x03)`. It accepts a `KVM_CAP_*` integer and returns 0 if the capability is absent or a positive integer if it is present — the exact positive value is capability-defined; for some it is just 1, for others it carries useful information like a count or size. A subset of capabilities can also be queried on the VM fd, if `KVM_CAP_CHECK_EXTENSION_VM` is supported, giving per-VM answers rather than per-system ones.

Firecracker on x86_64 checks exactly 14 capabilities at startup, referred to internally as `DEFAULT_CAPABILITIES`: `KVM_CAP_IRQCHIP`, `KVM_CAP_IOEVENTFD`, `KVM_CAP_IRQFD`, `KVM_CAP_USER_MEMORY`, `KVM_CAP_SET_TSS_ADDR`, `KVM_CAP_PIT2`, `KVM_CAP_PIT_STATE2`, `KVM_CAP_ADJUST_CLOCK`, `KVM_CAP_DEBUGREGS`, `KVM_CAP_MP_STATE`, `KVM_CAP_VCPU_EVENTS`, `KVM_CAP_XCRS`, `KVM_CAP_XSAVE`, and `KVM_CAP_EXT_CPUID`. It checks `KVM_CAP_XSAVE2` separately at runtime to decide whether to use the dynamically-sized XSAVE buffer path. A missing required capability is a fatal error; Firecracker never falls back to working without it.

The capabilities used directly in this chapter are `KVM_CAP_USER_MEMORY` (value 3), which gates `KVM_SET_USER_MEMORY_REGION`, and `KVM_CAP_NR_MEMSLOTS` (value 10), which returns the maximum number of memory slots the kernel supports. Query `KVM_CAP_NR_MEMSLOTS` at runtime; do not hardcode a slot limit.

Creating a VM

`KVM_CREATE_VM` is `_IO(0xAE, 0x01)`, issued on the system fd. It takes a machine-type argument; pass `0` for the default x86 machine type. It returns a new VM file descriptor.

```
int vmfd = ioctl(kvmfd, KVM_CREATE_VM, 0);
```

The returned VM fd has no vCPUs and no memory. Neither defaults to anything useful; both must be configured before the first `KVM_RUN` call. An empty VM is not an error — it is a valid but uninitialized state.

One non-obvious failure mode: on a heavily loaded machine, `KVM_CREATE_VM` can return `-1` with `errno = EINTR`. This is intentional. The kernel path calls `mm_take_all_locks()`, which is CPU-intensive and checks for pending signals. Firecracker handles this by retrying up to five times with exponential backoff starting at 1 µs, doubling on each attempt (1 µs, 2 µs, 4 µs, 8 µs, 16 µs). A VMM that treats `EINTR` from `KVM_CREATE_VM` as a fatal error will occasionally fail on healthy hosts under load.

Mapping Guest Physical Memory

The guest needs physical memory before it can run. KVM's model for guest memory is deliberately minimal: the VMM allocates host memory however it chooses — `mmap`, `malloc`, a `memfd` — and then registers the result with KVM via `KVM_SET_USER_MEMORY_REGION`. KVM does not manage guest memory allocation; it manages the *mapping* between host virtual addresses and guest physical addresses.

> ***Note:*** *These operations require that `/dev/kvm` is open and a VM fd exists. The program running the following ioctls must have read/write access to `/dev/kvm`, which on most Linux distributions means membership in the `kvm` group or root privilege. The backing `mmap` must remain valid for the entire lifetime of the VM; premature deallocation is undefined behavior with consequences the kernel cannot catch.*

`KVM_SET_USER_MEMORY_REGION` is `_IOW(0xAE, 0x46, struct kvm_userspace_memory_region)`, issued on the VM fd. The struct, from `include/uapi/linux/kvm.h`:

```
struct kvm_userspace_memory_region {
    __u32 slot;              /* bits 0-15: slot index;
                                bits 16-31: address space id
                                (requires KVM_CAP_MULTI_ADDRESS_SPACE) */
    __u32 flags;
    __u64 guest_phys_addr;   /* base guest physical address */
    __u64 memory_size;       /* bytes; 0 deletes the slot */
    __u64 userspace_addr;    /* host virtual address of backing memory */
};
```

`guest_phys_addr` is the base of the range in the guest's physical address space. `userspace_addr` is the host virtual address where the backing memory lives. The terminology is worth pausing on: the guest believes it is accessing physical memory at `guest_phys_addr`. The host sees an ordinary virtual address range. EPT provides the hardware translation between those two views, with the kernel building EPT entries that map guest-physical addresses to the host-physical pages underlying `userspace_addr`. An EPT violation — a guest access to a guest-physical address the EPT does not yet map — faults in through the kernel's EPT violation handler, without surfacing to the VMM userspace process at all, for addresses that the registered slots do cover. An access outside all registered slots produces `KVM_EXIT_MMIO` or, in recent kernels, `KVM_EXIT_MEMORY_FAULT`.

The `flags` field has three defined bits:

Flag	Bit	Meaning
`KVM_MEM_LOG_DIRTY_PAGES`	`1UL << 0`	Enable dirty page tracking for live migration
`KVM_MEM_READONLY`	`1UL << 1`	Read-only slot (guest writes produce a fault)
`KVM_MEM_GUEST_MEMFD`	`1UL << 2`	Backed by a guest memfd (TDX / AMD SNP only)

Firecracker assigns slot IDs via an atomic counter (`next_kvm_slot: AtomicU32`), sets `flags = KVM_MEM_LOG_DIRTY_PAGES` when dirty-page tracking is active for a snapshot, and passes `flags = 0` otherwise. Calling the ioctl with an existing `slot` number replaces the prior registration in-place. Passing `memory_size = 0` deletes the slot. For huge-page-backed memory the lower 21 bits of `guest_phys_addr` and `userspace_addr` should match so that EPT huge-page entries can be built without splitting the host's huge pages.

A newer `KVM_SET_USER_MEMORY_REGION2` (`_IOW(0xAE, 0x49, struct kvm_userspace_memory_region2)`) extends the struct with a `guest_memfd` field for confidential VMs running under Intel TDX or AMD SEV-SNP, where guest memory must be isolated from the host even at the hypervisor level. For standard Firecracker use, the original ioctl is sufficient.

Creating a vCPU

With memory mapped, the next step is a vCPU. `KVM_CREATE_VCPU` is `_IO(0xAE, 0x41)`, issued on the VM fd. It takes a `vcpu_id` integer that acts as the APIC ID on x86:

```
int vcpufd = ioctl(vmfd, KVM_CREATE_VCPU, 0);
```

Valid IDs are in `[0, max_vcpu_id)` where `max_vcpu_id` comes from `KVM_CHECK_EXTENSION(KVM_CAP_MAX_VCPU_ID)` (capability 128). Firecracker uses a 0-based index for `vcpu_id` and assigns each vCPU its own OS thread. For a two-vCPU guest, the host sees threads named `fc_vcpu 0` and `fc_vcpu 1`, each blocked in `KVM_RUN` whenever the guest is executing.

The kvm_run Shared Region

Communication between the running guest and userspace happens through a shared memory region mapped onto the vCPU fd. Its size is not a compile-time constant — the kernel appends additional buffers after the `struct kvm_run` header, including I/O port data and coalesced MMIO pages — so the VMM must query it first:

```
int mmap_size = ioctl(kvmfd, KVM_GET_VCPU_MMAP_SIZE, NULL);
struct kvm_run *run = mmap(NULL, mmap_size,
                           PROT_READ | PROT_WRITE, MAP_SHARED,
                           vcpufd, 0);
```

`KVM_GET_VCPU_MMAP_SIZE` is `_IO(0xAE, 0x04)`, a system ioctl. Using `sizeof(struct kvm_run)` as the mmap size is wrong; it truncates the region the kernel needs beyond the struct.

The layout of `struct kvm_run` from `include/uapi/linux/kvm.h`:

```
struct kvm_run {
    /* IN: written by userspace before KVM_RUN */
    __u8  request_interrupt_window;
    __u8  immediate_exit;
    __u8  padding1[6];

    /* OUT: written by kernel on exit */
    __u32 exit_reason;
    __u8  ready_for_interrupt_injection;
    __u8  if_flag;
    __u16 flags;

    /* SHARED: valid across all exits */
    __u64 cr8;
    __u64 apic_base;

    /* exit-specific data (anonymous union) */
    union { ... };

    /* optional sync-regs optimization */
    __u64 kvm_valid_regs;
    __u64 kvm_dirty_regs;
    union {
        struct kvm_sync_regs regs;
        char padding[2048];   /* SYNC_REGS_SIZE_BYTES */
    } s;
};
```

The two IN fields are written by userspace to control the next `KVM_RUN` call. `immediate_exit` set to `1` from a different thread causes `KVM_RUN` to return `-1` with `errno = EINTR`; `exit_reason` is stale in that case and must not be read. Firecracker uses this to pause a vCPU for device-model events: another thread sets `immediate_exit = 1`, checks for `EINTR`, then clears the flag. Note that `KVM_EXIT_INTR` (10) is a separate mechanism — it appears as `exit_reason` on a normal zero-return from `KVM_RUN` when a signal is pending in certain configurations, unrelated to the `-1` / `EINTR` error path.
`request_interrupt_window` set to `1` asks KVM to exit with `KVM_EXIT_IRQ_WINDOW_OPEN` (7) as soon as the vCPU's interrupt flag is set and a hardware interrupt could be injected, which is the mechanism for delivering queued interrupts when the guest had them masked.

`exit_reason` is written by the kernel on exit. Everything in the anonymous union that follows it is exit-specific and is only valid for the indicated exit code.

KVM_RUN

`KVM_RUN` is `_IO(0xAE, 0x80)`, issued on the vCPU fd. No argument is passed; all communication goes through the mmap'd `struct kvm_run`.

```
int ret = ioctl(vcpufd, KVM_RUN, NULL);
```

The return value encodes outcome:

Return	errno	Meaning
0	—	Normal exit; read `exit_reason` and loop
-1	`EINTR`	An unmasked signal interrupted the vCPU
-1	`ENOEXEC`	vCPU not yet initialized
-1	`EFAULT` or `EHWPOISON`	`KVM_EXIT_MEMORY_FAULT` (the one exit that sets errno rather than exit_reason)

The ordinary path returns 0. `exit_reason` then holds the reason code and the union holds the associated data. A return of `-1` with `EINTR` means a signal arrived; the VMM should check for pending signals, handle them, and call `KVM_RUN` again. `KVM_EXIT_MEMORY_FAULT` (value 39) is the exception to the exit_reason convention: when a guest access faults on a memory region for which no slot is registered, `KVM_RUN` returns `-1` with `errno = EFAULT` or `EHWPOISON`, not the usual `0` with `exit_reason = 39`.

KVM handles many exits internally without returning to userspace at all: EPT violations for pages within registered slots, APIC accesses with an in-kernel irqchip, and MSR reads and writes for MSRs KVM manages itself. The exits that surface to userspace are the ones the VMM must handle: I/O port accesses, MMIO accesses to unregistered regions, HLT, shutdown, and fail-entry. The set of in-kernel exits versus userspace exits is determined by the capabilities and irqchip configuration established before the first `KVM_RUN` call.

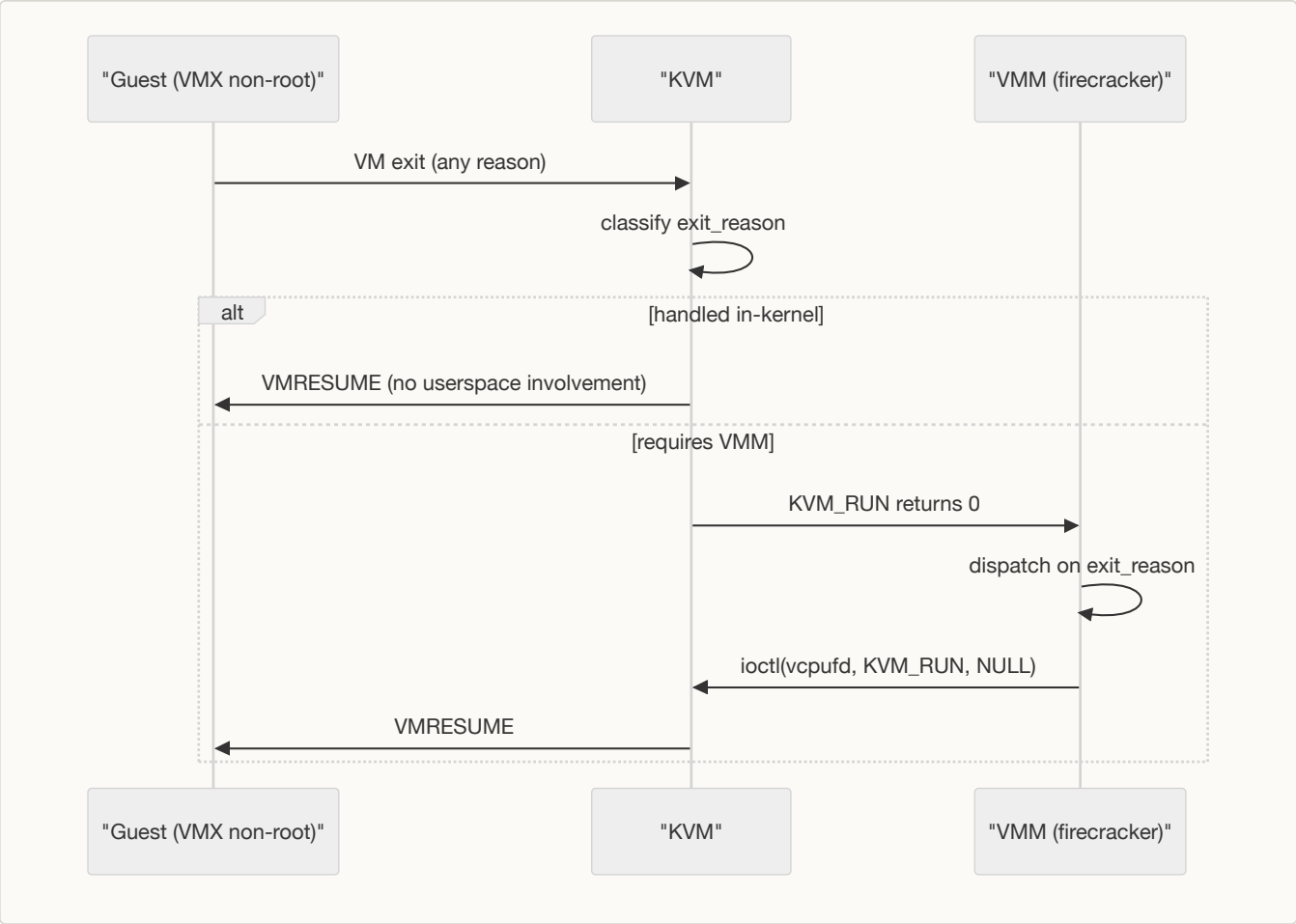
The Run Loop

The canonical C run loop, following the structure of the LWN example:

```
while (1) {
    ioctl(vcpufd, KVM_RUN, NULL);
    switch (run->exit_reason) {
    case KVM_EXIT_HLT:
        /* guest executed HLT with nothing queued */
        return 0;
    case KVM_EXIT_IO:
        /* data at (char *)run + run->io.data_offset */
        putchar(*(((char *)run) + run->io.data_offset));
        break;
    case KVM_EXIT_FAIL_ENTRY:
        errx(1, "hardware_entry_failure_reason = 0x%llx",
             run->fail_entry.hardware_entry_failure_reason);
    case KVM_EXIT_INTERNAL_ERROR:
        errx(1, "suberror = 0x%x", run->internal.suberror);
    default:
        errx(1, "unexpected exit reason: %d", run->exit_reason);
    }
}
```

Firecracker's run loop, in `src/vmm/src/vstate/vcpu.rs`, follows the same logic but with Rust's enum dispatch. The vCPU's `run_emulation()` function checks `kvm_run.immediate_exit == 1` before calling `fd.run()`, so that a preemption request from another thread is honored without an unnecessary `KVM_RUN` call. On `EINTR` it clears `immediate_exit` and returns `VcpuEmulation::Interrupted`. The exit dispatch in `handle_kvm_exit()` routes `MmioRead` and `MmioWrite` to the MMIO bus, `IoIn` and `IoOut` to the port I/O bus, treats `FailEntry` and `InternalError` as `VcpuError::FaultyKvmExit`, interprets `SystemEvent` with type `KVM_SYSTEM_EVENT_SHUTDOWN` (1) or `KVM_SYSTEM_EVENT_RESET` (2) as `VcpuEmulation::Stopped`, and returns `VcpuError::UnhandledKvmExit` for anything unrecognized. Device model dispatch for MMIO and port I/O happens synchronously in the vCPU thread; there is no separate I/O thread for those paths.

The sequence diagram below shows the two paths a VM exit can take:



Exit Reason Codes

The full enumeration in `include/uapi/linux/kvm.h` currently runs from `KVM_EXIT_UNKNOWN` (0) through `KVM_EXIT_SNP_REQ_CERTS` (43). For x86_64 microVM workloads, the exits that actually matter are:

Value	Constant	What triggered it
2	`KVM_EXIT_IO`	Guest `IN` or `OUT` instruction
5	`KVM_EXIT_HLT`	Guest `HLT` with no pending interrupt
6	`KVM_EXIT_MMIO`	Guest access to an MMIO address
7	`KVM_EXIT_IRQ_WINDOW_OPEN`	Interrupt window requested and now open
8	`KVM_EXIT_SHUTDOWN`	Guest triple-fault or CPU reset
9	`KVM_EXIT_FAIL_ENTRY`	Hardware refused VM entry
10	`KVM_EXIT_INTR`	Signal pending; normal zero-return exit
16	`KVM_EXIT_NMI`	NMI delivered to guest (x86, not handled by Firecracker)
17	`KVM_EXIT_INTERNAL_ERROR`	KVM internal error
24	`KVM_EXIT_SYSTEM_EVENT`	Guest requested shutdown or reset

Values 13–15 are S390/PowerPC-specific. Value 16 (`KVM_EXIT_NMI`) is x86 but Firecracker routes it to `VcpuError::UnhandledKvmExit` . Values 18–23, 25–27, and 28–43 cover S390, PowerPC, ARM, RISC-V, Xen, LoongArch, TDX, and AMD SNP exits outside the Firecracker scope.

The Exit-Specific Union

Each exit reason with userspace-visible data populates one arm of the anonymous union in `struct kvm_run` . Four arms are worth examining in detail.

KVM_EXIT_IO (2)

```
struct {
    __u8  direction;     /* KVM_EXIT_IO_IN = 0, KVM_EXIT_IO_OUT = 1 */
    __u8  size;          /* 1, 2, or 4 bytes per operation */
    __u16 port;          /* I/O port number */
    __u32 count;         /* number of operations (REP prefix) */
    __u64 data_offset;   /* offset from start of kvm_run to data buffer */
} io;
```

The data is not embedded in the union. It lives at `(char *)run + run->io.data_offset` , which is exactly why the mmap region is larger than `sizeof(struct kvm_run)` — the kernel places the I/O data in the overhang. Total bytes transferred are `io.count * io.size` . On a COM1 write at port `0x3f8` , `io.port = 0x3f8` , `io.direction = KVM_EXIT_IO_OUT` , `io.size = 1` , `io.count = 1` , and the character lives at `((char *)run) + run->io.data_offset` .

KVM_EXIT_MMIO (6)

```
struct {
    __u64 phys_addr;   /* guest physical address */
    __u8  data[8];     /* data bytes (embedded; max 8 bytes) */
    __u32 len;         /* 1, 2, 4, or 8 */
    __u8  is_write;    /* 0 = read, 1 = write */
} mmio;
```

Unlike `KVM_EXIT_IO`, the MMIO data is embedded directly in the struct at `run->mmio.data[0]`. There is no `data_offset` indirection. For a read, the VMM writes the response into `data` before calling `KVM_RUN` again; KVM will relay it to the guest.

KVM_EXIT_FAIL_ENTRY (9)

```
struct {
    __u64 hardware_entry_failure_reason;  /* VMX/SVM hardware error code */
    __u32 cpu;                            /* physical CPU on which the failure occurred */
} fail_entry;
```

`KVM_EXIT_FAIL_ENTRY` means the hardware declined to enter VMX non-root mode. The `hardware_entry_failure_reason` is a VMX VM-entry failure qualification or an SVM equivalent. This exit is almost always the result of incorrect segment descriptor setup, misaligned page tables, or an invalid VMCS state. It is not recoverable; the VMM should log the reason code and tear down the VM.

KVM_EXIT_INTERNAL_ERROR (17)

```
struct {
    __u32 suberror;   /* see below */
    __u32 ndata;      /* count of valid entries in data[] */
    __u64 data[16];
} internal;
```

`suberror` takes four defined values: `KVM_INTERNAL_ERROR_EMULATION` (1, the kernel failed to emulate an instruction), `KVM_INTERNAL_ERROR_SIMUL_EX` (2, simultaneous exceptions), `KVM_INTERNAL_ERROR_DELIVERY_EV` (3, event delivery error), and `KVM_INTERNAL_ERROR_UNEXPECTED_EXIT_REASON` (4). Like `KVM_EXIT_FAIL_ENTRY`, this exit is not recoverable in normal operation. Firecracker treats it as `VcpuError::FaultyKvmExit` and halts the VM.

KVM_EXIT_SYSTEM_EVENT (24)

```
struct {
    __u32 type;
    __u32 ndata;
    union { __u64 data[16]; };
} system_event;
```

`type` carries `KVM_SYSTEM_EVENT_SHUTDOWN` (1), `KVM_SYSTEM_EVENT_RESET` (2), `KVM_SYSTEM_EVENT_CRASH` (3), `KVM_SYSTEM_EVENT_WAKEUP` (4), `KVM_SYSTEM_EVENT_SUSPEND` (5), or `KVM_SYSTEM_EVENT_SEV_TERM` (6). This is the path a well-behaved guest kernel uses when it calls `reboot(2)` or shuts down cleanly; `KVM_EXIT_SHUTDOWN` (8) is the fault path when the guest triple-faults. Firecracker maps both to `VcpuEmulation::Stopped` and performs an orderly teardown.

Reading and Writing Guest Registers

Before the first `KVM_RUN` call, the VMM must initialize the vCPU register state. The default state after `KVM_CREATE_VCPU` leaves the vCPU pointed at the x86 reset vector — CS base at the top of the 4 GiB address space, RIP at `0xFFF0` — which is appropriate for BIOS-style boot but wrong for direct kernel boot. Firecracker skips the BIOS entirely and sets registers to the Linux 64-bit boot protocol entry state before the first run.

KVM_GET_REGS and KVM_SET_REGS

`KVM_GET_REGS` is `_IOR(0xAE, 0x81, struct kvm_regs)`. `KVM_SET_REGS` is `_IOW(0xAE, 0x82, struct kvm_regs)`. Both are vCPU ioctls. On x86, from `arch/x86/include/uapi/asm/kvm.h`:

```
struct kvm_regs {
    __u64 rax, rbx, rcx, rdx;
    __u64 rsi, rdi, rsp, rbp;
    __u64 r8,  r9,  r10, r11;
    __u64 r12, r13, r14, r15;
    __u64 rip, rflags;
};
```

`rflags` must be initialized to at least `0x2`. Bit 1 is architecturally reserved-must-be-1; a VM will fail to enter VMX non-root mode without it, and the failure surfaces as `KVM_EXIT_FAIL_ENTRY`.

Firecracker's `setup_regs()` in `src/vmm/src/arch/x86_64/regs.rs` writes two configurations depending on boot protocol:

```
Linux 64-bit boot protocol
  rflags = 0x2
  rip    = kernel_entry_addr
  rsp    = BOOT_STACK_POINTER
  rbp    = BOOT_STACK_POINTER
  rsi    = ZERO_PAGE_START   // pointer to boot-params zero page
```

```
PVH boot entry point
  rflags = 0x2
  rip    = entry_addr
  rbx    = PVH_INFO_START    // address of PVH start_info structure
```

KVM_GET_SREGS and KVM_SET_SREGS

The general-purpose registers in `kvm_regs` cover computation but not the CPU's operating mode. Segment registers, descriptor-table registers, control registers, and EFER live in a separate ioctl pair. `KVM_GET_SREGS` is `_IOR(0xAE, 0x83, struct kvm_sregs)`. `KVM_SET_SREGS` is `_IOW(0xAE, 0x84, struct kvm_sregs)`.

```
struct kvm_sregs {
    struct kvm_segment cs, ds, es, fs, gs, ss;
    struct kvm_segment tr, ldt;
    struct kvm_dtable gdt, idt;
    __u64 cr0, cr2, cr3, cr4, cr8;
    __u64 efer;
    __u64 apic_base;
    __u64 interrupt_bitmap[(KVM_NR_INTERRUPTS + 63) / 64]; /* 4 x __u64 */
};
```

`KVM_NR_INTERRUPTS` is 256, so `interrupt_bitmap` is four 64-bit words — a bitmap of pending external interrupts, one bit per IRQ vector 0–255.

`struct kvm_segment` maps directly to x86 descriptor fields:

```
struct kvm_segment {
    __u64 base;
    __u32 limit;
    __u16 selector;
    __u8  type;       /* 4-bit x86 segment type */
    __u8  present, dpl, db, s, l, g, avl;
    __u8  unusable;
    __u8  padding;
};
```

`dpl` is the descriptor privilege level (0–3), `db` is the default operand size, `s` distinguishes system (0) from code/data (1) descriptors, `l` marks a 64-bit code segment, `g` is the granularity bit, and `avl` is available for OS use.

The CS fix-up that every minimal KVM example needs: a freshly created vCPU's CS still has its base pointing to the reset vector at the top of the 4 GiB address space. Before setting RIP to an arbitrary guest physical address, the VMM must redirect CS. The LWN example does exactly this:

```
sregs.cs.base = 0;
sregs.cs.selector = 0;
```

Without this, a vCPU with `rip = 0x1000` and CS base still at `0xFFFF0000` would execute at effective address `0xFFFF1000`, not `0x1000`. The fix is two fields, not one.

For Firecracker's 64-bit long-mode setup, `setup_sregs()` works through a longer sequence: (1) call `KVM_GET_SREGS` to fetch the vCPU's current state, (2) write a 4-entry GDT at guest physical address `BOOT_GDT_OFFSET = 0x500` and a stub IDT at `BOOT_IDT_OFFSET = 0x520`, (3) set `cr0` with `PE | PG` (`0x1 | 0x80000000`) to enter protected mode with paging enabled, `cr4` with `PAE` (`0x20`) to enable physical address extension, and `efer` with `LME | LMA` (`0x100 | 0x400`) to activate 64-bit long mode, (4) install page tables with PML4 at GPA `0x9000`, PDPTE at `0xa000`, and PDE at `0xb000`, and (5) commit with `KVM_SET_SREGS`. The PVH path sets `cr0` with `PE | ET` (`0x1 | 0x10`) instead, reflecting the different boot entry contract. The guest vCPU enters its first `KVM_RUN` already in 64-bit long mode, with valid page tables and a stack — there is no mode-switch sequence during boot at all.

The Ioctl Number Reference

All ioctls discussed in this chapter:

Ioctl	Macro	Scope
`KVM_GET_API_VERSION`	`_IO(0xAE, 0x00)`	System
`KVM_CREATE_VM`	`_IO(0xAE, 0x01)`	System
`KVM_CHECK_EXTENSION`	`_IO(0xAE, 0x03)`	System / VM
`KVM_GET_VCPU_MMAP_SIZE`	`_IO(0xAE, 0x04)`	System
`KVM_CREATE_VCPU`	`_IO(0xAE, 0x41)`	VM
`KVM_SET_USER_MEMORY_REGION`	`_IOW(0xAE, 0x46, struct kvm_userspace_memory_region)`	VM
`KVM_RUN`	`_IO(0xAE, 0x80)`	vCPU
`KVM_GET_REGS`	`_IOR(0xAE, 0x81, struct kvm_regs)`	vCPU
`KVM_SET_REGS`	`_IOW(0xAE, 0x82, struct kvm_regs)`	vCPU
`KVM_GET_SREGS`	`_IOR(0xAE, 0x83, struct kvm_sregs)`	vCPU
`KVM_SET_SREGS`	`_IOW(0xAE, 0x84, struct kvm_sregs)`	vCPU

For ioctls built with `_IOW` or `_IOR`, the numeric ioctl value includes the struct size encoded in bits 16–29. Compute it from the macro rather than hardcoding the integer.

What the API Surface Reveals

The three-scope ioctl model is not arbitrary. It enforces a lifecycle: a system fd cannot issue vCPU commands, and a VM fd cannot run code. Crossing a scope boundary is an immediate `ENOTTY`. This structure also means the kernel can perform per-scope privilege checks — a process that holds only a VM fd forwarded over a Unix socket cannot enumerate other VMs on the system.

The shared `kvm_run` region is the design's most interesting tradeoff. KVM could have passed exit information in ioctl arguments, but shared memory allows the `immediate_exit` and `request_interrupt_window` fields to be written from a different thread without any syscall. The cost is that the VMM must be careful about concurrent access to the region — one field is in, one is out, and the kernel writes `exit_reason` only when `KVM_RUN` returns. In practice the per-vCPU thread model Firecracker uses makes the concurrency straightforward: the vCPU thread owns the region except for the two flags another thread might set.

The next chapter works through what happens when the guest accesses an address outside any registered memory slot, and how EPT and NPT mediate the full address translation from guest-virtual to host-physical through two independent page table walks.

Sources And Further Reading

- KVM API kernel documentation (canonical reference for all ioctls, struct definitions, and exit reason codes): https://docs.kernel.org/virt/kvm/api.html
- `include/uapi/linux/kvm.h` (exit reason constants, `struct kvm_run`, `struct kvm_userspace_memory_region`, capability values): https://github.com/torvalds/linux/blob/master/include/uapi/linux/kvm.h
- `arch/x86/include/uapi/asm/kvm.h` (`struct kvm_regs`, `struct kvm_sregs`, `struct kvm_segment`, `struct kvm_dtable`): https://github.com/torvalds/linux/blob/master/arch/x86/include/uapi/asm/kvm.h
- LWN "Using the KVM API" (Josh Triplett, September 29, 2015) — the canonical minimal KVM example in C, covering the three-fd hierarchy, CS fix-up, and the `KVM_EXIT_IO` / `KVM_EXIT_HLT` loop: https://lwn.net/Articles/658511/
- Firecracker vCPU run loop (`vstate/vcpu.rs`) — `run_emulation()`, `handle_kvm_exit()`, `immediate_exit` handling, and `VcpuEmulation` variants: https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/vstate/vcpu.rs
- Firecracker VM creation (`vstate/vm.rs`) — `KVM_CREATE_VM` retry logic and capability check invocation: https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/vstate/vm.rs
- Firecracker memory region setup (`vstate/memory.rs`) — `KVM_SET_USER_MEMORY_REGION` slot assignment and dirty-tracking flag: https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/vstate/memory.rs
- Firecracker x86_64 register setup (`arch/x86_64/regs.rs`) — `setup_regs()` and `setup_sregs()` for both Linux boot protocol and PVH: https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/arch/x86_64/regs.rs
- Firecracker x86_64 capability checks (`arch/x86_64/kvm.rs`) — `DEFAULT_CAPABILITIES` list and `KVM_CAP_XSAVE2` runtime check: https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/arch/x86_64/kvm.rs
- rust-vmm kvm-ioctls crate (Rust wrappers for the KVM ioctl surface, including `VcpuFd::run()` and the `VcpuExit` enum): https://github.com/rust-vmm/kvm
- kvm-ioctls API documentation: https://docs.rs/kvm-ioctls/latest/kvm_ioctls/

Chapter 6: Guest Memory And Two-Dimensional Paging

The previous chapter registered guest memory with `KVM_SET_USER_MEMORY_REGION` and moved on. What it did not explain is what "guest physical address" actually means, or what the hardware does when a guest instruction fetches from one. The guest OS believes it is running on a machine whose RAM starts at address zero. The VMM process is running on a host OS that has no idea a guest exists. Somehow a load instruction issued inside the VM must reach host physical DRAM — and on a busy host that memory might be at physical address `0x2_4e3f_0000`, nothing close to what the guest sees. This chapter is about how that bridging works, what data structures implement it, and what it costs.

The answer is two-dimensional paging: the guest's page tables translate guest virtual to guest physical, and a second hardware page-table tree translates guest physical to host physical. A TLB miss has to walk both trees. Intel calls its implementation EPT (Extended Page Tables); AMD calls its implementation NPT (Nested Page Tables), marketed as RVI (Rapid Virtualization Indexing). Before those technologies existed, hypervisors had to maintain shadow page tables that merged both translations into one, at a maintenance cost so severe that the hardware acceleration provided speed-ups of up to 48% on MMU-intensive benchmarks and up to 600% on MMU-intensive microbenchmarks.

Four Address Spaces, Not Two

The KVM MMU documentation (`Documentation/virt/kvm/x86/mmu.rst`) defines four distinct address spaces that coexist in a virtualized x86-64 system. GPA is not HVA; confusing the two is the most common error in VMM memory code.

Symbol	Name	Controlled by
GVA	Guest Virtual Address	Guest OS — CR3-rooted 4-level page tables inside the VM
GPA	Guest Physical Address	VMM — KVM memory slots and the EPT/NPT structure
HVA	Host Virtual Address	VMM process `mmap` — an ordinary pointer in the Firecracker address space
HPA	Host Physical Address	Host OS page tables — where DRAM actually is

The translation chain in EPT/NPT mode is:

```
GVA --(guest page tables, CR3)--> GPA
GPA --(KVM memslot lookup)--> HVA
HVA --(host page tables)--> HPA
```

The instinct to equate GPA with HPA, or GPA with HVA, is the most common mental model error when reading VMM source code. GPA is not HVA. The guest believes its RAM starts at GPA zero; the VMM allocated the backing memory at some HVA like `0x7f3a80000000`. The two are related only by the slot registration. Similarly, GPA is not HPA — the host kernel placed the physical backing wherever it saw fit when the VMM called `mmap`. None of these three equalities hold in production. The hardware's job, through EPT/NPT, is to make the distinction invisible to guest code.

Memory Slots

KVM's model for guest memory is built around **memory slots**: named, numbered regions that declare "guest physical addresses from `guest_phys_addr` to `guest_phys_addr + memory_size` are backed by host virtual memory starting at `userspace_addr`." A slot is not the memory itself; it is a mapping declaration. The VMM supplies the backing memory by any means it chooses, and KVM uses the slot to build the EPT/NPT entries that make the translation fast.

KVM_SET_USER_MEMORY_REGION

`KVM_SET_USER_MEMORY_REGION` is `_IOW(KVMIO, 0x46, struct kvm_userspace_memory_region)`, a VM ioctl issued on the VM file descriptor. It requires capability `KVM_CAP_USER_MEMORY`. The struct, from `include/uapi/linux/kvm.h`:

```
struct kvm_userspace_memory_region {
    __u32 slot;            /* bits 0-15: slot index; bits 16-31: address space ID */
    __u32 flags;
    __u64 guest_phys_addr; /* GPA base of this slot */
    __u64 memory_size;     /* bytes; 0 = delete this slot */
    __u64 userspace_addr;  /* HVA: host virtual address of backing memory */
};
```

The slot index in bits 0–15 is the identifier KVM uses to distinguish slots. Bits 16–31 carry the address space ID, used when `KVM_CAP_MULTI_ADDRESS_SPACE` is available — irrelevant for most VMMs, which operate in address space zero. Calling the ioctl with an existing slot number replaces that slot's mapping in-place. Passing `memory_size = 0` deletes the slot. Slots must not overlap in guest physical address space; the kernel enforces this and returns `-EINVAL` on a conflict.

The `flags` field has three defined bits:

Flag	Bit	Meaning
`KVM_MEM_LOG_DIRTY_PAGES`	`0x1`	KVM maintains a dirty bitmap; retrieve with `KVM_GET_DIRTY_LOG`
`KVM_MEM_READONLY`	`0x2`	Guest writes produce `KVM_EXIT_MMIO`; requires `KVM_CAP_READONLY_MEM`
`KVM_MEM_GUEST_MEMFD`	`0x4`	Backed by a guest memfd; only valid in `KVM_SET_USER_MEMORY_REGION2`

On ARM64, a write to a `KVM_MEM_READONLY` slot injects an abort into the guest rather than generating `KVM_EXIT_MMIO` — a behavioral difference worth knowing if the code targets multiple architectures.

Slot Counts

The kernel on x86-64 supports 509 user-accessible slots (plus 3 internal slots, for a total `KVM_MEM_SLOTS_NUM` of 512). That limit was raised from 125 to 509 in a 2014 patch to support 256 memory hotplug slots plus 253 device slots. ARM64 was raised from 32 to 508 in a later patch; the 32-slot limit had constrained PCI passthrough device counts. Query the runtime limit via `KVM_CHECK_EXTENSION(KVM_CAP_NR_MEMSLOTS)` rather than hardcoding 509 — `KVM_CAP_NR_MEMSLOTS` is capability value 10, the same value checked in `KVM_CHECK_EXTENSION(KVM_CAP_NR_MEMSLOTS)` from chapter 5.

Internally, the kernel stores memory slots in `struct kvm_memory_slot` (from `include/linux/kvm_host.h`), which holds `base_gfn` (the GPA base right-shifted by `PAGE_SHIFT`), `npages`, `userspace_addr`, `dirty_bitmap`, `id`, and `as_id`. The slots are indexed by a red-black tree keyed by guest frame number and a hash table keyed by slot ID, with two sets maintained for lockless readers — an active set and an inactive set swapped via a generation counter on each update.

The Extended Variant

A newer `KVM_SET_USER_MEMORY_REGION2` (`_IOW(KVMIO, 0x49, struct kvm_userspace_memory_region2)`) extends the struct with `guest_memfd_offset` and `guest_memfd` fields. It requires capabilities `KVM_CAP_GUEST_MEMFD` and `KVM_CAP_USER_MEMORY2`. The extension exists for confidential computing — Intel TDX and AMD SEV-SNP — where guest memory must be isolated from the host even at the hypervisor level. Standard Firecracker deployments use the original ioctl.

The VMM mmap Pattern

The canonical sequence for allocating and registering guest RAM, following the LWN "Using the KVM API" article:

> ***Note:*** *The program issuing these calls must have read/write access to `/dev/kvm`. On most Linux distributions that means membership in the `kvm` group or root privilege. The `mmap` backing must remain live for the entire lifetime of the VM. Unmapping it while the VM is running is undefined behavior; the kernel cannot intercept the deallocation.*

```
/* Step 1: allocate anonymous pages in the host process */
void *mem = mmap(NULL, size,
                 PROT_READ | PROT_WRITE,
                 MAP_PRIVATE | MAP_ANONYMOUS,  /* MAP_SHARED also works; Linux ignores the
flag for anonymous mappings */
                 -1, 0);

/* Step 2: declare the GPA->HVA mapping to KVM */
struct kvm_userspace_memory_region region = {
    .slot            = 0,
    .guest_phys_addr = 0x1000,
    .memory_size     = size,
    .userspace_addr  = (uint64_t)mem,
};
ioctl(vmfd, KVM_SET_USER_MEMORY_REGION, &region);
```

`userspace_addr` is an HVA — a pointer in the VMM process's address space. The kernel has not touched the physical pages yet; the host kernel maps them lazily on first access. KVM records the slot's HVA range and uses it to build the EPT/NPT leaf entries that will resolve GPA to HPA when the guest faults in each page.

The backing can be anonymous memory (`MAP_ANONYMOUS`), file-backed memory (`memfd` or `hugetlbfs`), or device memory. The kernel recommends — not requires — that bits 20:0 of `guest_phys_addr` and `userspace_addr` be identical. The reason is alignment: both addresses must have the same offset within a 2 MiB boundary (2^21 bytes) for a host huge page to back a guest huge page without sub-page remapping. Violating this alignment is not an error, but it silently prevents huge-page EPT/NPT entries.

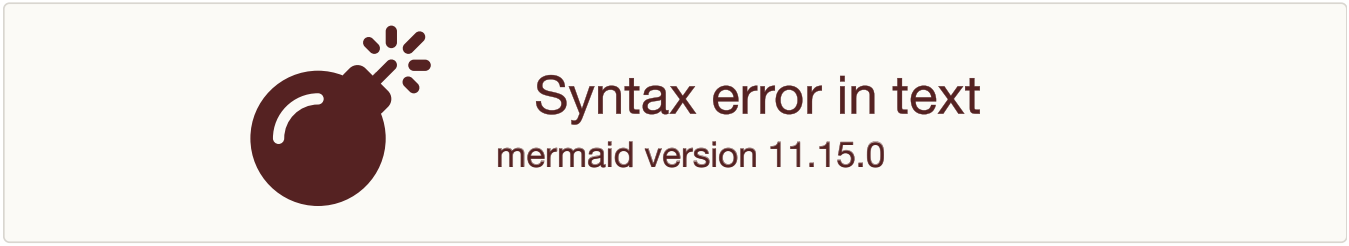
Firecracker's Slot Layout

Firecracker uses the `vm-memory` crate from the `rust-vmm` project, with two principal types:

```
pub type GuestRegionMmap = vm_memory::GuestRegionMmap<Option<AtomicBitmap>>;
pub type GuestMemoryMmap = vm_memory::GuestRegionCollection<GuestRegionMmapExt>;
```

`GuestRegionMmapExt` wraps `GuestRegionMmap` and adds `region_type` (Dram or Hotpluggable), `slot_from` (starting KVM slot number), `slot_size` (uniform byte size per KVM slot), and `plugged` (a `Mutex<BitVec>` tracking which sub-slots are currently active). The `AtomicBitmap` type tracks dirty pages at page granularity using atomic operations, so multiple vCPU threads can mark pages concurrently without a lock. When dirty tracking is disabled, `AtomicBitmap` is `None`.

The slot registration is a `From` implementation in `src/vmm/src/vstate/memory.rs`:



This `From` impl is the entire gap between Rust types and the raw ioctl struct. `KVM_MEM_LOG_DIRTY_PAGES` is set if and only if the `AtomicBitmap` is present — a clean expression of the dirty-tracking flag as a type-level choice.

Firecracker's GPA layout on x86-64, from `src/vmm/src/arch/x86_64/layout.rs`:

Region	GPA Start	Notes
Low RAM	`0x0`	Up to the 32-bit MMIO hole
EBDA / system data	`0x9fc00`	MPTable, ACPI tables
Kernel load	`0x0010_0000` (1 MiB)	`HIMEM_START`
32-bit MMIO gap	`0xC000_0000` – `0xFFFF_FFFF`	Device BARs, LAPIC, IOAPIC
RAM above 4 GiB	`0x1_0000_0000` +	Second slot for guests larger than ~3 GiB
64-bit MMIO gap	`0x40_0000_0000`	`MMIO64_MEM_START = 256 << 30`

The MMIO hole between roughly 3.25 GiB and 4 GiB is reserved for device configuration. Guest RAM that would otherwise land in that range is placed above 4 GiB in a second KVM slot. This means Firecracker uses at most two RAM memory slots on x86-64.

Backing mode is determined at construction time in `MmapRegion`: `anonymous()` produces `MAP_PRIVATE | MAP_ANONYMOUS` memory (with optional hugepages), `memfd_backed()` produces `MAP_SHARED` memory via a `memfd` file descriptor, and `snapshot_file()` uses `MAP_PRIVATE` from a file for snapshot restore.

Two-Dimensional Paging: EPT And NPT

Shadow Paging: The Problem It Solved and Created

Before EPT and NPT existed, KVM used shadow page tables. A `struct kvm_mmu_page` held 512 shadow PTEs (SPTEs) that mapped GVA directly to HPA, effectively collapsing the two-level translation into one. That sounds efficient, but the maintenance cost was severe. KVM had to write-protect all guest page tables so it could intercept modifications; every guest CR3 load, every `INVLPG`, and every page-table write triggered a VM exit so KVM could rebuild or invalidate the corresponding shadow entries. The KVM

MMU documentation notes that in EPT mode "neither invlpg nor CR3 loads and stores cause a vmexit in EPT mode, and kvm_set_cr3 is hardly ever called" — describing, by contrast, how intrusive shadow paging was.

Shadow paging was also the source of a central MMU lock that serialized all vCPU threads on page-fault handling, a design that broke down catastrophically at scale. That lock is what motivated the TDP MMU rewrite discussed below.

The 24-Access Worst Case

A TLB miss under two-level nested paging on x86-64 requires up to 24 memory accesses in the worst case. The derivation: the guest has 4-level page tables (PML4 → PDPT → PD → PT); each of those four guest-page-table entries is itself a GPA that must be resolved through the 4-level nested page table, costing 4 accesses per guest-table walk level plus 1 for the nested PML4 root. That gives 4 × 5 = 20 accesses for the guest walk, plus 4 more to translate the final GPA to HPA: 24 total. This is the cold-TLB worst case with no EPT or NPT TLB entries populated. In steady state the hardware TLBs cache the composed translations and most accesses cost nothing beyond a normal TLB hit.

Intel EPT

EPT was introduced in the Nehalem microarchitecture — the first Intel Core i-series, around 2008. The "unrestricted guest" mode, which allows a guest to run in real mode without shadow paging, requires EPT and was added in the subsequent Westmere generation.

Enabling EPT. EPT is activated through VMCS Secondary Processor-Based VM-Execution Controls, encoding `0x401E` (`SECONDARY_VM_EXEC_CONTROL`, confirmed in `arch/x86/include/asm/vmx.h`). Bit 1 of that field is "Enable EPT." Setting it to 1 activates hardware two-level paging for that VM.

EPT Pointer. Once EPT is enabled, the hardware needs to know where the root of the EPT paging structure lives. VMCS field `EPT_POINTER` (encoding `0x201A`, confirmed in `arch/x86/include/asm/vmx.h`) is written via `VMWRITE` to supply that root. The EPTP bit layout:

Bits	Meaning
2:0	EPT paging-structure memory type (0 = UC, 6 = WB; WB is normal)
5:3	Page-walk length minus 1 (3 = 4-level EPT, the current standard)
6	Enable accessed and dirty flags in EPT entries (requires CPU support; absent before Haswell)
11:7	Reserved, must be zero
51:12	Physical address of EPT PML4 table
63:52	Reserved

A 5-level EPT (PML5) was added for 57-bit guest physical addresses; bits 5:3 would be 4 for 5-level.

EPT leaf PTE fields. Each EPT entry is 8 bytes. The leaf PTE bits that matter most:

Bit	Meaning
0	Read permission
1	Write permission
2	Execute permission (supervisor mode)
5:3	EPT memory type (6 = WB)
8	Accessed flag (set by hardware)
9	Dirty flag (set by hardware on write; leaf entries only)
51:12	Host physical page frame address

EPT violations and misconfigurations. An EPT violation exits when a guest access lacks sufficient EPT permission — for example, a write to a read-only EPT entry. Exit reason `EXIT_REASON_EPT_VIOLATION = 48` (from `arch/x86/include/uapi/asm/vmx.h`). An EPT misconfiguration exits when an EPT entry has an illegal format, such as a non-leaf entry with write permission set but read permission clear. Exit reason `EXIT_REASON_EPT_MISCONFIG = 49`. KVM uses EPT violations deliberately for MMIO interception: MMIO ranges are left unmapped in the EPT, so a guest access generates an EPT violation that KVM handles as `KVM_EXIT_MMIO` back to the VMM, without any explicit MMIO range registration in EPT.

```
flowchart TD
  A["Guest memory access (GPA)"]
  F{"Entry format valid?\n(e.g. write=1 but read=0 is illegal)"}
  G["EPT misconfiguration → EXIT_REASON_EPT_MISCONFIG (49)"]
  B{"Entry present?\n(read | write | execute != 0)"}
  D{"Permission sufficient\nfor this access?"}
  E["EPT violation → EXIT_REASON_EPT_VIOLATION (48)"]
  C["Hardware resolves GPA→HPA (no exit)"]

  A --> F
  F -- no --> G
  F -- yes --> B
  B -- no --> E
  B -- yes --> D
  D -- no --> E
  D -- yes --> C
```

EPTP switching. VM function 0 allows VMX non-root software to switch EPT roots without a full VM exit, by indexing into a hypervisor-controlled list of 512 8-byte EPTP entries via ECX. VMCS VM-function controls live at encoding `0x2018`; the EPTP-list address lives at `0x2024`. This is a niche optimization for

workloads that need to quickly present different physical memory views to a guest.

AMD NPT

AMD introduced nested paging in 3rd-generation Opteron (codename Barcelona, 2007), one year before Intel's Nehalem. AMD's marketing name is Rapid Virtualization Indexing; the engineering name is NPT. Performance gains over shadow paging: VMware research measured up to 42%; Red Hat OLTP testing showed approximately 2× throughput improvement.

Enabling NPT. AMD's VM control block is the VMCB, a structure distinct from Intel's VMCS. Bit 0 of the `np_enable` field (VMCB control area offset `0x90`) activates nested paging when the `VMRUN` instruction is issued.

nCR3. The nested page table root is held in `nCR3` (Nested CR3), a 64-bit field at VMCB control area offset `0xB0` (confirmed in FreeBSD's `sys/amd64/vmm/amd/vmcb.h` as `VMCB_OFF_NPT_BASE`). It holds a host physical address — the HPA of the top-level NPT paging structure. The guest's ordinary CR3 (`gCR3`) holds a GPA of the guest's own page-table root. Both pointers are active simultaneously; this is the fundamental asymmetry. The hardware consults `gCR3` for GVA→GPA and `nCR3` for GPA→HPA.

ASID. Each guest is assigned an Address Space Identifier at VMCB control area offset `0x58` (`VMCB_OFF_ASID`), so the hardware can tag TLB entries per-guest and avoid full TLB flushes on VM entry and exit.

Nested page faults. A nested page fault generates SVM exit code `SVM_EXIT_NPF = 0x400` (from `arch/x86/include/uapi/asm/svm.h`). KVM module parameters `kvm-amd.npt=0` and `kvm-intel.ept=0` disable NPT and EPT respectively at module load time; the default for both is 1 (enabled for 64-bit and 32-bit PAE mode).

VMCB clean bits. Bit 4 of the VMCB clean-bits field (offset `0xC0`) is `VMCB_CACHE_NP`, signaling that the nested paging fields including `nCR3` are clean and need not be reloaded from VMCB on VM entry. This caching reduces the cost of rapid `VMRUN` calls on nested-paging paths.

The TDP MMU

KVM's TDP MMU (`arch/x86/kvm/mmu/tdp_mmu.c`) is a reimplementation of the KVM MMU designed specifically for EPT/NPT. It eliminates the reverse mapping (rmap) data structure that shadow paging required. Shadow paging needed rmaps to find every SPTE that mapped a given host physical page — necessary for write-protection maintenance. TDP page tables are per-VM and map GPA directly to HPA without indirection through GVA contexts, so no rmap is needed. Eliminating rmap removed the central MMU lock that serialized all vCPU threads.

When the TDP MMU was introduced as a 22-patch series in September 2020, Google measured an 89% reduction in demand-paging test duration on 416-vCPU VMs; previously 98% of time was spent waiting for the MMU lock. The TDP MMU enabled live migration of 416-vCPU, 12 TiB VMs that had been

impractical with the legacy MMU. **The TDP MMU became the default for x86-64 KVM in Linux 5.15.** The legacy shadow MMU remains as a fallback when EPT/NPT is unavailable.

In TDP mode, the SPTE role has `role.base.direct = true` (direct GPA→HPA mapping), with `role.base.cr0_wp` and `role.base.efer_nx` unconditionally set to true — unlike shadow paging, where they reflect actual guest CPU state. KVM supports 4 KiB (level-1 SPTE), 2 MiB (level-2), and 1 GiB (level-3) EPT/NPT entries. A large SPTE requires that the host supports the page size, that the guest PTE maps an equivalent range, that no write-protected pages exist in the range, and that the entire range falls within a single memory slot.

Dirty-Page Tracking

Two use cases drive dirty-page tracking: live migration (which must replay every write the guest makes after the first full copy) and snapshot diffing (which records only pages changed since the last snapshot). KVM offers two interfaces for the same underlying data: a per-slot bitmap and a per-vCPU ring buffer. They are mutually exclusive for a given VM.

The Legacy Bitmap Interface

Setting `KVM_MEM_LOG_DIRTY_PAGES` (`0x1`) in `kvm_userspace_memory_region.flags` instructs the kernel to maintain a dirty bitmap for that slot — one bit per 4 KiB page, bit 0 corresponding to the first page. The bitmap lives in `kvm_memory_slot.dirty_bitmap`. The kernel also sets EPT/NPT entries in that slot to read-only, so a guest write generates a fault that sets the bit and restores write permission. This write-protection overhead is why enabling dirty-page tracking forces 4 KiB EPT/NPT granularity even when the host uses 2 MiB huge pages: a huge-page EPT entry cannot be write-protected at 4 KiB sub-page granularity.

`KVM_GET_DIRTY_LOG` (`_IOW(KVMIO, 0x42, struct kvm_dirty_log)`) retrieves the bitmap for one slot:

```
struct kvm_dirty_log {
    __u32 slot;
    __u32 padding1;
    union {
        void __user *dirty_bitmap;
        __u64        padding2;
    };
};
```

By default, the kernel clears dirty bits atomically before the ioctl returns. `KVM_CAP_MANUAL_DIRTY_LOG_PROTECT2` (capability value 168) defers that clearing to a subsequent `KVM_CLEAR_DIRTY_LOG` call. `KVM_CLEAR_DIRTY_LOG` (`_IOWR(KVMIO, 0xc0, struct kvm_clear_dirty_log)`) adds `__u32 num_pages` and `__u64 first_page` fields, enabling partial range clearing rather than whole-slot clearing — useful for large slots where clearing the entire bitmap stalls the guest.

`KVM_CAP_MANUAL_DIRTY_LOG_PROTECT2` supports two sub-flags: `KVM_DIRTY_LOG_MANUAL_PROTECT_ENABLE (1 << 0)` and `KVM_DIRTY_LOG_INITIALLY_SET (1 << 1)`. When `INITIALLY_SET` is active, the bitmap starts all-ones, treating all pages as initially dirty. A VMM can then use `KVM_CLEAR_DIRTY_LOG` to write-unprotect pages in chunks, spreading the VM-exit storm of re-enabling protection across time rather than concentrating it at the moment tracking is enabled. `KVM_DIRTY_LOG_INITIALLY_SET` is incompatible with the dirty ring interface.

The Dirty Ring Interface

`KVM_CAP_DIRTY_LOG_RING` (capability value 192) enables the dirty ring. The ring is a per-vCPU mmap'd region, separate from `kvm_run`, containing `struct kvm_dirty_gfn` entries:

```
struct kvm_dirty_gfn {
    __u32 flags;  /* KVM_DIRTY_GFN_F_DIRTY = (1<<0), KVM_DIRTY_GFN_F_RESET = (1<<1) */
    __u32 slot;
    __u64 offset; /* page offset within slot */
};
```

The state machine is: `flags = 0` means the entry is invalid (empty slot); `flags = 1` (DIRTY set) means the kernel has recorded a dirty GFN; `flags = 3` (both DIRTY and RESET set) means userspace has read the entry and is requesting reset. Userspace reads ring entries without any ioctl. After harvesting entries, it issues `KVM_RESET_DIRTY_RINGS` (`_IO(KVMIO, 0xc7)`) to re-write-protect the harvested pages.

The ring has a genuine trade-off. Under an 800 MB/s random-write rate with a 24 GiB guest, dirty ring required approximately 73 seconds to complete live migration versus approximately 55 seconds for dirty bitmap. At very high dirty rates, the per-page write-protection overhead of the ring can exceed the cost of the bitmap's bulk clearing, making the bitmap faster. The right choice depends on workload: the ring shines when dirty pages are scattered and sparse; the bitmap wins under sustained high write pressure.

Firecracker's Dirty Tracking

Firecracker's `MachineConfig` (in `src/vmm/src/vmm_config/machine_config.rs`) has a `track_dirty_pages: bool` field, default `false`. When `true`, Firecracker sets `KVM_MEM_LOG_DIRTY_PAGES` on all memory slots, and each `GuestRegionMmapExt` receives a `Some(AtomicBitmap)` rather than `None`.

The snapshot flow in `src/vmm/src/vmm_config/snapshot.rs` uses `store_dirty_bitmap()` to read KVM's dirty log and merge it into the internal `AtomicBitmap`. `dump_dirty()` then iterates 64-bit words of the merged bitmap, seeking past clean regions using sparse-file semantics, and writes only dirty 4 KiB pages to the diff snapshot file. After a diff snapshot, Firecracker resets the dirty bitmap to baseline the next diff.

Without `track_dirty_pages`, Firecracker falls back to `mincore(2)` to identify resident pages. This mode requires swap to be disabled: a page swapped out appears as not-in-core and would be silently omitted from the snapshot. The trade-off is that `mincore` produces no write overhead at runtime, while

`track_dirty_pages` introduces the write-protection overhead described above and forces 4 KiB granularity even when the host uses hugepages. Diff snapshots are currently in developer preview.

Userfaultfd for Snapshot Resume

When restoring a VM from a snapshot, the VMM must repopulate guest memory without stalling the guest for the full restore to complete. Firecracker supports two modes, controlled by `LoadSnapshotParams.mem_backend.backend_type`: `File` (blocking read-back) and `Uffd` (demand-paged via userfaultfd).

In the `Uffd` path, a separate userspace process receives the userfaultfd file descriptor over a Unix domain socket and responds to `UFFD_EVENT_PAGEFAULT` by issuing `UFFDIO_COPY` to populate individual pages on demand as the guest touches them. On Linux 5.10, the userfaultfd object is created via the `userfaultfd(2)` syscall; on Linux 6.1 and later it is created via `/dev/userfaultfd`. When the virtio-balloon deflates during a UFFD-backed resume, `madvise(MADV_DONTNEED)` triggers `UFFD_EVENT_REMOVE`, and the page handler must zero those pages rather than reloading from the snapshot file — a subtle interaction between two separately designed subsystems that Firecracker's documentation explicitly warns about.

Memory Ballooning

Ballooning is the mechanism by which the host can reclaim memory from a running guest without stopping it. The guest OS voluntarily surrenders pages through a device driver, and the VMM releases the backing host memory. The protocol is virtio.

The Virtio Balloon Device

The virtio balloon device has device ID 5 (OASIS virtio 1.2 spec §5.5.1). It uses up to four virtqueues: index 0 (inflate queue — guest passes PFNs of pages to surrender), index 1 (deflate queue — guest passes PFNs of pages to reclaim), index 2 (stats queue, enabled with `VIRTIO_BALLOON_F_STATS_VQ`), and index 3 (free-page hint queue, enabled with `VIRTIO_BALLOON_F_FREE_PAGE_HINT`). The protocol is asymmetric: the host signals how many pages it wants by writing `num_pages` into the `virtio_balloon_config` struct; the guest driver responds at its own pace by inflating or deflating through the queues. The host cannot force the guest to respond promptly.

The feature bits that define balloon behavior (from `include/uapi/linux/virtio_balloon.h`):

Bit	Constant	Meaning
0	`VIRTIO_BALLOON_F_MUST_TELL_HOST`	Guest must notify host before reusing deflated pages
1	`VIRTIO_BALLOON_F_STATS_VQ`	Enables stats virtqueue (index 2)
2	`VIRTIO_BALLOON_F_DEFLATE_ON_OOM`	Guest deflates balloon instead of invoking OOM killer
3	`VIRTIO_BALLOON_F_FREE_PAGE_HINT`	Guest reports free pages to host (index 3)
4	`VIRTIO_BALLOON_F_PAGE_POISON`	Guest reports page-poison value via `poison_val` config field
5	`VIRTIO_BALLOON_F_REPORTING`	Guest reports free pages via reporting queue for host to reclaim

The config struct carries two fields visible to the driver: `__le32 num_pages` (how many pages the host wants in the balloon) and `__le32 actual` (how many are currently held). The `stats` queue exchanges `struct virtio_balloon_stat` entries — tag-value pairs, packed, 10 bytes each — with 16 defined tags as of Linux 6.12, including swap-in/out counts, major/minor faults, free and total memory, OOM kills, and direct and async reclaim statistics.

Firecracker's Balloon

Firecracker exposes the balloon through its REST API:

- Pre-boot: `PUT /balloon` with `{"amount_mib": N, "deflate_on_oom": bool, "stats_polling_interval_s": N}`
- Runtime: `PATCH /balloon` to adjust target size and polling interval
- `GET /balloon/statistics` to read the stats virtqueue values

Firecracker supports three virtio-balloon feature bits: `VIRTIO_BALLOON_F_DEFLATE_ON_OOM` (bit 2), `VIRTIO_BALLOON_F_FREE_PAGE_HINT` (bit 3, developer preview), and `VIRTIO_BALLOON_F_REPORTING` (bit 5).

When the guest inflates the balloon — surrendering pages — Firecracker issues `madvise(MADV_DONTNEED)` on the corresponding HVA range. This releases the underlying host physical pages back to the host kernel and reduces the Firecracker process RSS. On the next access, the host kernel zero-fills the page (anonymous `MAP_PRIVATE` semantics), preventing cross-VM data leakage. The Firecracker documentation notes a critical asymmetry: the host can set `num_pages` to request balloon inflation, but the actual surrender rate is governed by the guest kernel's balloon driver. An operator cannot count on the balloon responding within any particular time bound. The documentation's warning is worth repeating directly: ensure the host is prepared to handle a situation in which the Firecracker process uses all of the memory it was given at boot, even if the balloon was used to restrict guest memory.

Oversubscription

Firecracker's design document states that microVMs can oversubscribe host CPU and memory; the degree is controlled by the operator. No built-in hard cap is enforced. The `mmap` call for guest RAM succeeds as long as virtual address space is available; host physical pages are committed lazily on first access. An operator running 100 Firecracker processes, each with 512 MiB of guest RAM, is not necessarily using 50 GiB of host RAM — only the pages the guests have actually touched are physically backed.

Firecracker's production host setup guide (`docs/prod-host-setup.md`) mandates two settings that define the oversubscription envelope:

Disable swap. `/proc/swaps` must be empty. The reason is not performance but security: guest memory swapped to host storage creates data remanence — a page that was part of a guest's heap appears on disk, accessible to the host operator and potentially recoverable after the VM is destroyed. With swap disabled, the only way memory pressure resolves is through the balloon.

Disable KSM. `echo 0 > /sys/kernel/mm/ksm/run` . KSM (Kernel Same-page Merging) deduplicates pages with identical content across processes, saving physical RAM. The security cost is a timing side channel: by measuring how long certain memory operations take, a process can determine which pages are shared with another process — leaking information about memory access patterns across VM boundaries. Disabling KSM removes this channel entirely.

> ***Note:*** *Both changes require root and affect the entire host, not just the Firecracker process. Disable swap and KSM before starting any Firecracker production workload, not inline with the VMM's startup sequence.*

With swap and KSM disabled, virtio-balloon is the only host-side mechanism for reclaiming memory from running VMs. Cgroup `memory.limit_in_bytes` (or the v2 equivalent `memory.max`) provides a hard ceiling on how much memory a single Firecracker process can consume, which is the primary per-VM isolation tool. The operator's oversubscription ratio is the ratio of total `memory_size` across all memory slots across all VMs to the host's physical RAM, minus a safety margin for the guest kernels' actual working sets.

The GPA→HPA chain built in this chapter — slots declaring the GVA→GPA mapping, EPT/NPT translating GPA→HPA in hardware, dirty tracking and ballooning managing the physical backing — is the axis that everything else in a VMM's memory subsystem rotates around. The next chapter turns from memory layout to device I/O: how a guest reaches storage and network without knowing it is virtualized.

Sources And Further Reading

- KVM API kernel documentation (canonical reference for `KVM_SET_USER_MEMORY_REGION` , dirty log ioctls, flags, capability values): https://docs.kernel.org/virt/kvm/api.html

- KVM MMU documentation (`Documentation/virt/kvm/x86/mmu.rst`) — address space definitions (GVA, GPA, HVA, HPA), shadow paging vs. TDP, SPTE levels: https://docs.kernel.org/virt/kvm/x86/mmu.html
- `include/uapi/linux/kvm.h` — ioctl encodings, `struct kvm_userspace_memory_region`, dirty log structs, dirty ring structs, capability constants: https://github.com/torvalds/linux/blob/master/include/uapi/linux/kvm.h
- `include/linux/kvm_host.h` — internal `struct kvm_memory_slot` (red-black tree, hash table, `dirty_bitmap` field): https://github.com/torvalds/linux/blob/master/include/linux/kvm_host.h
- `arch/x86/include/asm/vmx.h` — VMCS field encodings (`EPT_POINTER = 0x0000201A`, `SECONDARY_VM_EXEC_CONTROL = 0x0000401E`): https://github.com/torvalds/linux/blob/master/arch/x86/include/asm/vmx.h
- `arch/x86/include/uapi/asm/vmx.h` — VMX exit reason codes (`EXIT_REASON_EPT_VIOLATION = 48`, `EXIT_REASON_EPT_MISCONFIG = 49`): https://github.com/torvalds/linux/blob/master/arch/x86/include/uapi/asm/vmx.h
- `arch/x86/include/uapi/asm/svm.h` — SVM exit codes (`SVM_EXIT_NPF = 0x400`): https://github.com/torvalds/linux/blob/master/arch/x86/include/uapi/asm/svm.h
- `include/uapi/linux/virtio_balloon.h` — virtio-balloon feature bits and stat tag definitions: https://github.com/torvalds/linux/blob/master/include/uapi/linux/virtio_balloon.h
- FreeBSD `sys/amd64/vmm/amd/vmcb.h` — AMD VMCB layout (`VMCB_OFF_NPT_BASE = 0xB0`, `VMCB_OFF_ASID = 0x58`, `VMCB_CACHE_NP` bit 4): https://github.com/freebsd/freebsd-src/blob/master/sys/amd64/vmm/amd/vmcb.h
- rust-vmm kvm-bindings (`src/x86_64/bindings.rs`) — `KVM_MEM_LOG_DIRTY_PAGES = 0x1`, `KVM_MEM_READONLY = 0x2`: https://github.com/rust-vmm/kvm-bindings/blob/main/src/x86_64/bindings.rs
- ia32-doc machine-readable Intel SDM extract (VMCS field encodings): https://github.com/wbenny/ia32-doc/blob/master/yaml/Intel/VMX/VMCS.yml
- Firecracker memory types and KVM slot registration (`src/vmm/src/vstate/memory.rs`): https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/vstate/memory.rs
- Firecracker x86-64 GPA layout (`src/vmm/src/arch/x86_64/layout.rs`): https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/arch/x86_64/layout.rs
- Firecracker `track_dirty_pages` field (`src/vmm/src/vmm_config/machine_config.rs`): https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/vmm_config/machine_config.rs
- Firecracker snapshot config (`src/vmm/src/vmm_config/snapshot.rs`): https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/vmm_config/snapshot.rs
- Firecracker snapshot support documentation (diff snapshots, `mincore` fallback, dirty-tracking constraints): https://github.com/firecracker-

microvm/firecracker/blob/main/docs/snapshotting/snapshot-support.md

- Firecracker page-fault handling on snapshot resume (UFFD backend, `UFFD_EVENT_PAGEFAULT`, `UFFDIO_COPY`, `UFFD_EVENT_REMOVE`): https://github.com/firecracker-microvm/firecracker/blob/main/docs/snapshotting/handling-page-faults-on-snapshot-resume.md
- Firecracker hugepages documentation (dirty-tracking incompatibility with huge pages): https://github.com/firecracker-microvm/firecracker/blob/main/docs/hugepages.md
- Firecracker ballooning documentation (REST API, `MADV_DONTNEED`, supported feature bits): https://github.com/firecracker-microvm/firecracker/blob/main/docs/ballooning.md
- Firecracker production host setup guide (no swap, no KSM, cgroup memory limits): https://github.com/firecracker-microvm/firecracker/blob/main/docs/prod-host-setup.md
- Firecracker design document (oversubscription policy and design goals): https://github.com/firecracker-microvm/firecracker/blob/main/docs/design.md
- OASIS virtio 1.2 specification §5.5 (balloon device ID 5, virtqueues, feature bits, config struct): https://docs.oasis-open.org/virtio/virtio/v1.2/virtio-v1.2.html
- vm-memory crate bitmap/AtomicBitmap documentation: https://docs.rs/vm-memory/latest/vm_memory/bitmap/index.html
- LWN "Using the KVM API" (Josh Triplett) — canonical `mmap` + `KVM_SET_USER_MEMORY_REGION` pattern: https://lwn.net/Articles/658511/
- LWN TDP MMU introduction (September 2020) — 89% demand-paging improvement, 416-vCPU VMs, no-rmap design: https://lwn.net/Articles/832835/
- LWN dirty ring performance data — 800 MB/s random-write rate, 24 GiB guest, 73 s ring vs. 55 s bitmap: https://lwn.net/Articles/833784/
- Phoronix: TDP MMU made default in Linux 5.15: https://www.phoronix.com/news/Linux-5.15-KVM
- KVM x86 memslot increase patch (125 → 509 user slots, 2014): https://patchwork.kernel.org/patch/5244591/
- ARM64 memslot increase patch (32 → 508 user slots): https://patchwork.kernel.org/project/linux-arm-kernel/patch/1486538141-30627-3-git-send-email-linucherian@gmail.com/
- Wikipedia: Second Level Address Translation (EPT Nehalem introduction, NPT Barcelona introduction, 24-access derivation, performance gain figures): https://en.wikipedia.org/wiki/Second_Level_Address_Translation
- KVM memory overview (nCR3 vs. gCR3 distinction): https://www.linux-kvm.org/page/Memory
- ACRN hypervisor memory management (EPT violations, misconfigurations, and MMIO interception pattern): https://projectacrn.github.io/latest/developer-guides/hld/hv-memmgt.html
- `KVM_CAP_MANUAL_DIRTY_LOG_PROTECT2` `INITIALLY_SET` details: https://patchwork.kernel.org/patch/11419191/
- Dirty ring and bitmap exclusivity, `INITIALLY_SET` incompatibility: https://lkml.kernel.org/kvm/20200331190000.659614-7-peterx@redhat.com/

Chapter 7: Virtual Interrupts And Time

A guest operating system expects to receive interrupts. The NIC driver expects the hardware to signal that a packet arrived. The block driver expects a completion notice. The timer subsystem expects the interrupt controller to fire on schedule. None of that hardware exists — the guest is running on emulated devices — yet the guest must receive interrupts that arrive with plausible timing and in the right order. Meanwhile, the guest's clock must advance at something close to wall-clock rate, survive live migration to a host whose TSC runs at a slightly different frequency, and not drift by seconds across a long-lived workload.

The interrupt and timekeeping problems share an ancestor. Both require the hypervisor to intercept hardware-level operations — writes to interrupt-controller registers, reads of the time-stamp counter — and either emulate them fully in kernel space or share a memory-mapped ABI with the guest that makes the emulation cheap enough to be invisible. Getting the first problem wrong produces missed interrupts, hangs, and stalled virtio queues. Getting the second produces drifting logs, failing TLS certificates, and confused application timing loops.

The Interrupt Architecture Problem

On real hardware, an interrupt follows a path that software rarely thinks about: a device asserts a line, the I/O APIC records it, the local APIC on the target CPU raises an interrupt request, and the CPU saves state and dispatches the handler. This path crosses several pieces of hardware the guest believes it owns — the local APIC register page at `0xFEE00000`, the I/O APIC at `0xFEC00000`, the legacy 8259 PIC pair at I/O ports `0x20` and `0xA0` — none of which exists in the guest's physical address space unless the hypervisor puts it there.

The hypervisor has three options. It can emulate the entire interrupt controller stack in userspace — every register write from the guest triggers a VM exit, the VMM updates its software model, and the VMM injects the interrupt on the next `KVM_RUN`. It can emulate the controllers inside the kernel, handling register accesses without returning to userspace. Or it can split the work: keep the local APIC in-kernel where injection is fast, but handle the legacy PIC and I/O APIC in userspace where the VMM can apply its own routing policy. Each choice makes different tradeoffs between latency, flexibility, and complexity.

In-Kernel Irqchip vs. Userspace

The canonical path for x86 microVMs is fully in-kernel. A single VM-level ioctl, `KVM_CREATE_IRQCHIP` (`_IO(KVMIO, 0x60)`), requires `KVM_CAP_IRQCHIP` (capability value 0) and creates three emulated controllers in one call: a master 8259 PIC (`KVM_IRQCHIP_PIC_MASTER = 0`), a slave 8259 PIC (`KVM_IRQCHIP_PIC_SLAVE = 1`), and an I/O APIC (`KVM_IRQCHIP_IOAPIC = 2`). Every vCPU created after this call gets an in-kernel local APIC. The state of any chip can be read or written later with `KVM_GET_IRQCHIP` and `KVM_SET_IRQCHIP`, which reference chips by these same ID constants.

On arm64, `KVM_CREATE_IRQCHIP` creates a GICv2 only. For a GICv3 — the interrupt controller on any recent Arm server or embedded SoC — userspace must instead call `KVM_CREATE_DEVICE` with `KVM_DEV_TYPE_ARM_VGIC_V3`. The kernel enforces that only one VGIC instance may exist per VM; GICv2 and GICv3 cannot coexist.

Split-irqchip mode, enabled by `KVM_CAP_SPLIT_IRQCHIP` (value 121), keeps the in-kernel local APIC but moves the legacy PIC, I/O APIC, and PIT to userspace. When a guest performs an EOI that would normally notify the I/O APIC, KVM surfaces this to userspace as `KVM_EXIT_IOAPIC_EOI` rather than handling it in-kernel. This is the configuration chosen by VMMs that want fine-grained control over routing while still keeping interrupt injection fast.

Without either `KVM_CREATE_IRQCHIP` or split mode, the VMM emulates every controller in userspace and injects interrupts via the vCPU ioctl `KVM_INTERRUPT`, which queues a single interrupt vector for delivery at the next VM entry. This pure-userspace path requires a round-trip to userspace for every interrupt and is too slow for production use; it exists for completeness and for VMMs that deliberately avoid the kernel irqchip.

Firecracker calls `KVM_CREATE_IRQCHIP` at VM creation on x86_64, using the fully in-kernel path for all three chips. On aarch64, it provisions a GICv3 via `KVM_CREATE_DEVICE` with `KVM_DEV_TYPE_ARM_VGIC_V3`, falling back to `KVM_DEV_TYPE_ARM_VGIC_V2` if the host kernel or hardware does not support GICv3.

The Local APIC In Detail

After `KVM_CREATE_IRQCHIP`, each vCPU's local APIC is emulated by KVM in `arch/x86/kvm/lapic.c`. The APIC register page is 4 KiB (`LAPIC_MMIO_LENGTH = 4096`), mapped at `0xFEE00000` in the guest's physical address space. Register accesses go through KVM's MMIO handler rather than to real hardware, so they are resolved in-kernel without a userspace round-trip.

The key registers and their offsets within the APIC page:

Register	Offset	Purpose
`APIC_ID`	`0x020`	APIC identifier
`APIC_LVR`	`0x030`	Version; KVM emulates `0x14`
`APIC_SPIV`	`0x0F0`	Spurious interrupt vector; bit `APIC_SPIV_APIC_ENABLED` arms the APIC
`APIC_ICR`	`0x300`	Interrupt Command Register, low 32 bits
`APIC_ICR2`	`0x310`	ICR high 32 bits — destination field
`APIC_LVT0`	`0x350`	LVT entry 0 (LINT0)
`APIC_LVT1`	`0x360`	LVT entry 1 (LINT1)

The Interrupt Request Register (IRR), In-Service Register (ISR), and Trigger-Mode Register (TMR) are each 256-bit bitmaps stored as eight 32-bit registers inside the `kvm_lapic_state.regs` page. When the guest writes the EOI register, KVM's handler calls `apic_find_highest_isr()`, clears the ISR bit for the current interrupt, recomputes the Processor Priority Register, and notifies the in-kernel I/O APIC via `kvm_ioapic_send_eoi()`. All of this happens inside the APIC MMIO handler without returning to userspace.

The LAPIC timer complicates things. Emulating a timer accurately requires the kernel to know when the next deadline fires, then absorb the jitter introduced by VM-exit and VM-entry latency. KVM applies a configurable timer advance: `LAPIC_TIMER_ADVANCE_NS_INIT = 1000` ns, capped at `LAPIC_TIMER_ADVANCE_NS_MAX = 5000` ns. The kernel fires the host-side timer slightly early, then busy-waits in a tight loop to hit the precise deadline, spending less than one microsecond of CPU time per timer event on a quiet system.

The full LAPIC state can be saved and restored across migration with `KVM_GET_LAPIC` (ioctl `0x8e`) and `KVM_SET_LAPIC` (ioctl `0x8f`). Both operate on `struct kvm_lapic_state { char regs[KVM_APIC_REG_SIZE]; }` where `KVM_APIC_REG_SIZE = 0x400` (1024 bytes).

x2APIC

The original xAPIC mode uses 8-bit APIC IDs stored in MMIO registers. x2APIC extends this to 32-bit IDs and replaces the MMIO interface with MSR accesses in the range `0x800` – `0x8FF` — each MSR maps to an APIC register at offset `(msr - APIC_BASE_MSR) << 4` where `APIC_BASE_MSR = 0x800`. KVM emulates the full x2APIC MSR range, but doing so requires `KVM_CREATE_IRQCHIP` to have been called first; KVM does not support forwarding x2APIC MSR accesses to userspace.

Enabling the extended API requires `KVM_CAP_X2APIC_API` (value 129). When the `KVM_X2APIC_API_USE_32BIT_IDS` flag is set within this capability, KVM stores the full 32-bit x2APIC ID in bytes 32–35 of `kvm_lapic_state.regs`; xAPIC stores only an 8-bit ID in byte 35 (bits 31–24 of that word).

The In-Kernel I/O APIC

The KVM I/O APIC (`arch/x86/kvm/ioapic.c`) emulates exactly 24 input pins (`KVM_IOAPIC_NUM_PINS = 24`), matching the Intel 82093AA specification. The MMIO window is 256 bytes (`0x100`) at default base `0xFEC00000`. Like real hardware, the I/O APIC uses an indirect addressing scheme: a write to `IOAPIC_REG_SELECT` at offset `0x00` sets the internal register index; a subsequent read or write to `IOAPIC_REG_WINDOW` at offset `0x10` accesses the selected register.

Indirect register `0x00` is the ID (`IOAPICID`), `0x01` the version (`IOAPICVER`; KVM reports `IOAPIC_VERSION_ID = 0x11`), `0x02` the arbitration register (`IOAPICARB`). Redirection table entries start at index `0x10` (pin 0) and occupy two 32-bit words each, running through `0x3F` (pin 23). Each 64-

bit entry (`union kvm_ioapic_redirect_entry`) encodes the destination vector, delivery mode, destination APIC ID, trigger mode (edge or level), mask bit, and remote IRR flag.

GSI routing determines which controller receives each interrupt: GSIs 0–15 route to both the PIC and the I/O APIC (for compatibility with legacy software); GSIs 16–23 go to the I/O APIC only. The RTC IRQ, `RTC_GSI = 8` , routes through both.

The GSI Routing Table

Higher-level interrupt routing — from a device's logical signal to the right controller and pin — lives in a table the VMM manages with `KVM_SET_GSI_ROUTING` (`_IOW(KVMIO, 0x6a, struct kvm_irq_routing)`), gated by `KVM_CAP_IRQ_ROUTING` (value 25). Each call atomically replaces the entire table; there is no incremental-update path. A VMM that needs to add one route must rebuild and resubmit the full table.

Each entry in the table is a `struct kvm_irq_routing_entry` :

```
struct kvm_irq_routing_entry {
    __u32 gsi;
    __u32 type;   /* KVM_IRQ_ROUTING_IRQCHIP=1, KVM_IRQ_ROUTING_MSI=2,
                     KVM_IRQ_ROUTING_S390_ADAPTER=3, KVM_IRQ_ROUTING_HV_SINT=4,
                     KVM_IRQ_ROUTING_XEN_EVTCHN=5 */
    __u32 flags;
    __u32 pad;
    union {
        struct kvm_irq_routing_irqchip irqchip;
        struct kvm_irq_routing_msi     msi;
        /* ... */
    } u;
};
```

For MSI devices, `KVM_IRQ_ROUTING_MSI = 2` entries carry `address_lo` , `address_hi` , and `data` — the three fields that encode the destination APIC and vector in the MSI message format. Setting `KVM_MSI_VALID_DEVID` (bit 0 in `struct kvm_msi.flags`) passes a PCIe Requester ID via `devid` , which enables interrupt remapping hardware to associate the interrupt with a specific device (requires `KVM_CAP_MSI_DEVID = 131`).

On arm64, GSI routing applies to `KVM_IRQFD` bindings but does not apply to `KVM_IRQ_LINE` .

Firecracker builds its routing table by collecting all unmasked entries — one `KVM_IRQ_ROUTING_IRQCHIP` entry pointing to `KVM_IRQCHIP_IOAPIC` per device on x86_64, one entry with chip index 0 per device on aarch64, and one `KVM_IRQ_ROUTING_MSI` entry per MSI-capable device — and submits them with a single `set_gsi_routing()` call. Rebuilding on every change is acceptable because routing changes are rare and the atomicity guarantee is valuable.

irqfd: Interrupt Injection Without a VM Exit

The most important observation about interrupt delivery is that the fast path does not involve userspace at all. The mechanism that enables this is `irqfd`, introduced in Linux 2.6.32 (commit `721eecbf`, Gregory Haskins, Novell) and requiring `KVM_CAP_IRQFD` (value 32).

The underlying primitive is `eventfd(2)` (available since Linux 2.6.22): a file description backed by a 64-bit kernel counter. Writing 8 bytes adds to the counter; reading 8 bytes returns and resets it. The fd becomes `EPOLLIN`-readable the moment the counter is nonzero. KVM exploits the poll notification mechanism: during irqfd registration, `kvm_irqfd_assign()` calls `init_poll_funcptr()` and `vfs_poll()` on the eventfd file description, installing a custom waitqueue entry whose `irqfd_wakeup` function is called the instant the counter becomes nonzero — that is, the instant a device thread writes to the eventfd.

```
struct kvm_irqfd {
    __u32 fd;          /* eventfd file descriptor */
    __u32 gsi;         /* irqchip GSI / pin number */
    __u32 flags;
    __u32 resamplefd;  /* used only with KVM_IRQFD_FLAG_RESAMPLE */
    __u8  pad[16];
};
```

`KVM_IRQFD` is a VM ioctl: `_IOW(KVMIO, 0x76, struct kvm_irqfd)`. Setting `KVM_IRQFD_FLAG_DEASSIGN` in `flags` removes the binding (both `fd` and `gsi` must be provided). Setting `KVM_IRQFD_FLAG_RESAMPLE` (requires `KVM_CAP_IRQFD_RESAMPLE = 82`) switches to level-triggered mode: when the guest performs an EOI, KVM de-asserts the GSI and writes `resamplefd`, allowing the VMM to re-inject if the device still has work pending.

The path through the kernel (in `virt/kvm/eventfd.c`) avoids acquiring the `irqfds.lock` during the fast path to prevent deadlock; SRCU read locks protect the routing table instead. When `irqfd_wakeup` fires on `EPOLLIN`, it takes an SRCU read lock, reads the cached IRQ routing, and calls `kvm_arch_set_irq_inatomic()`. If that call returns `-EWOULDBLOCK` — the interrupt cannot be injected atomically at this moment, perhaps because the vCPU is not in a state that can receive it — the function schedules the `irqfd_inject` work item on a workqueue for deferred delivery. On `EPOLLHUP` (the eventfd was closed), `irqfd_deactivate()` removes the registration and queues `irqfd_shutdown`.

The result: a device thread writes 8 bytes to a file descriptor, and the guest receives an interrupt without the VMM process ever executing a line of code to handle it. No `KVM_RUN` return, no userspace round-trip, no scheduling decision — just a kernel callback and a VMCS update.

ioeventfd: Eliminating the Outbound Round-Trip

irqfd handles the host-to-guest direction. The guest-to-host direction — a guest writing to a virtqueue notify register to tell the host that work is ready — needs a mirror primitive. Without one, every guest MMIO write to a device register causes a `KVM_EXIT_MMIO` return from `KVM_RUN`, the VMM process wakes up, reads the address and value from `kvm_run.mmio`, dispatches to the right device handler, and re-enters the guest. That sequence costs on the order of 9 microseconds per notification.

`KVM_IOEVENTFD` (`_IOW(KVMIO, 0x89, struct kvm_ioeventfd)`, requiring `KVM_CAP_IOEVENTFD = 36`) was introduced in Linux 2.6.32 (commit `d34e6b17`, Gregory Haskins, August 2009) to eliminate that round-trip:

```
struct kvm_ioeventfd {
    __u64 datamatch;
    __u64 addr;    /* legal pio/mmio address */
    __u32 len;     /* 0, 1, 2, 4, or 8 bytes */
    __s32 fd;
    __u32 flags;
    __u8  pad[36];
};
```

The `flags` field controls matching behavior:

Flag	Meaning
`KVM_IOEVENTFD_FLAG_DATAMATCH`	Signal only if the written value matches `datamatch`
`KVM_IOEVENTFD_FLAG_PIO`	Target PIO address space instead of MMIO
`KVM_IOEVENTFD_FLAG_DEASSIGN`	Remove the binding
`KVM_IOEVENTFD_FLAG_VIRTIO_CCW_NOTIFY`	s390 virtio-ccw channel device

`KVM_CAP_IOEVENTFD_ANY_LENGTH` permits `len = 0` registrations that match regardless of write size.

The kernel fast-path in `virt/kvm/eventfd.c`: `kvm_assign_ioeventfd_idx()` registers the ioeventfd on KVM's MMIO, PIO, or VIRTIO_CCW bus via `kvm_io_bus_register_dev()`. When the guest executes a write to the registered address, KVM's exit handler calls `ioeventfd_write()`, which checks the address, the write length, and (if `KVM_IOEVENTFD_FLAG_DATAMATCH`) the written value via `ioeventfd_in_range()`. On a hit, it calls `eventfd_signal()` in-kernel and returns 0 — preventing the exit from propagating to userspace. A hardware-level VM exit still occurs (VMX must trap the write to unmapped MMIO), but the kernel services it without returning to the VMM process.

The patch commit message from August 2009 reported the performance effect:

Path	IOPS	Round-trip latency
QEMU MMIO baseline	110,000	9.09 µs
ioeventfd MMIO	200,100	5.00 µs
ioeventfd PIO	367,300	2.72 µs

The ioeventfd path recovers roughly 4 µs per notification by eliminating the userspace hop. For a workload with high virtqueue notification rates — sustained disk I/O or network traffic — that 4 µs per operation accumulates into a significant fraction of total CPU time.

How Firecracker Uses irqfd and ioeventfd

Firecracker registers one ioeventfd per virtqueue. The MMIO notify address is `device_base + NOTIFY_REG_OFFSET` where `NOTIFY_REG_OFFSET = 0x50` (defined in `src/vmm/src/devices/virtio/mod.rs`, matching the virtio MMIO specification). The datamatch value is the queue index `i`, so KVM signals the queue-`i` eventfd only when the guest writes `i` to the `QueueNotify` register — avoiding spurious signals when the guest notifies a different queue at the same address:

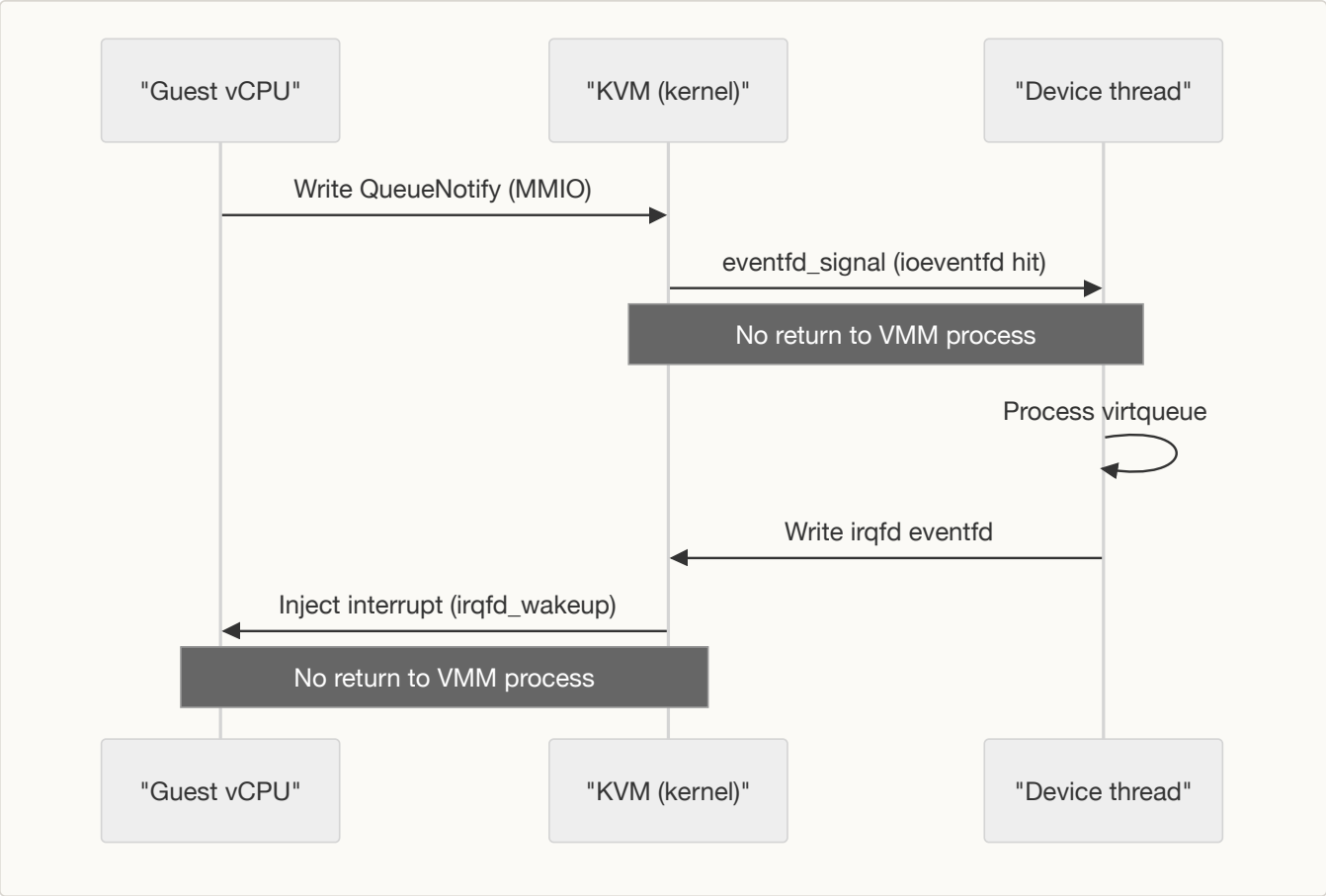
```
// src/vmm/src/device_manager/mmio.rs (simplified)
for (i, queue_evt) in locked_device.queue_events().iter().enumerate() {
    let io_addr = IoEventAddress::Mmio(
        device.resources.addr + u64::from(NOTIFY_REG_OFFSET),
    );
    vm.fd()
        .register_ioevent(queue_evt, &io_addr, u32::try_from(i).unwrap())
        .map_err(MmioError::RegisterIoEvent)?;
}
```

Each device's irqfd registration assigns a single GSI (allocated via `resource_allocator.allocate_gsi_legacy(1)`) and binds it to the device's interrupt eventfd:

```
vm.register_irq(&mmio_device.interrupt.irq_evt, gsi)
    .map_err(MmioError::RegisterIrqFd)?;
```

The virtio device thread polls the per-queue ioeventfd file descriptors via epoll; the interrupt eventfd is the irqfd that triggers a guest interrupt when the device signals completion. The MMIO slot for each virtio device is 4 KiB (`0x1000`).

The two mechanisms are complements:



The VMM process is not in the path for either the notification or the interrupt injection. It configured the bindings at setup time; the kernel and the device thread handle the fast path entirely.

Intel Posted Interrupt Processing

Even with irqfd, interrupt injection has a cost: when `irqfd_wakeup` fires and calls `kvm_arch_set_irq_inatomic()`, KVM must update the vCPU's interrupt state in the VMCS — which means the vCPU thread must either be in a state where the update is safe, or the injection must wait for the next VM entry. Intel VT-x includes a hardware mechanism that eliminates even this software step for the common case.

Posted interrupt processing was introduced on Ivy Bridge-EP and Haswell server processors in 2013. Two VMCS fields enable it: the **posted-interrupt notification vector**, a dedicated interrupt vector the CPU checks on incoming interrupts, and the **posted-interrupt descriptor address**, a pointer to a 64-byte **Posted-Interrupt Descriptor** (PID) in memory. All modifications to the PID must use locked read-modify-write instructions because the CPU and software may access it concurrently.

The PID layout:

Bits	Field	Meaning
255:0	PIR	256-bit bitmap; bit N indicates interrupt vector N is pending
256	ON	Outstanding Notification bit
511:257	—	Reserved

When an interrupt arrives while the vCPU is in VMX non-root mode and the interrupt vector matches the notification vector, the CPU does not exit. Instead it atomically clears the ON bit (bit 256), scans the PIR bitmap, and delivers all pending interrupts directly to the virtual APIC — without executing any VMM or KVM code. The cost of interrupt delivery when the vCPU is running is reduced to pure hardware time.

When the target vCPU is not currently scheduled, KVM sends an IPI carrying the notification vector to the physical CPU that will next run the vCPU. That CPU processes the PIR bits when it next enters VMX non-root mode for the vCPU, delivering the interrupt at entry rather than requiring an exit and re-entry cycle.

Enabling posted interrupts requires setting the "process posted interrupts" VM-execution control bit in the VMCS, configuring the notification vector and descriptor address, and ensuring `KVM_CAP_X2APIC_API` and related capabilities are in order. KVM manages this transparently when the hardware supports it; the VMM does not need to handle posted interrupts explicitly.

Paravirtual Interrupt Optimizations

Even with in-kernel APIC emulation, certain interrupt operations are expensive. An EOI write to the APIC at `0xFEE000B0` is an MMIO write that KVM must intercept and handle. On a guest processing thousands of interrupts per second, those EOI exits accumulate.

PV-EOI eliminates most of them. The guest writes `MSR_KVM_PV_EOI_EN = 0x4b564d04` with the low bit set and bits 63–2 holding a 4-byte-aligned guest physical address. KVM then sets bit 0 of the word at that address before injecting each interrupt. The guest's interrupt return path tests and clears that bit atomically; if the bit was set, the EOI is complete without any APIC MMIO write. Only when the bit is already clear — when multiple interrupt levels are active and the APIC needs to update the ISR — does the guest fall back to the MMIO EOI.

The paravirtual hypercall interface provides additional shortcuts for inter-processor interrupts:

Hypercall	Number	Description
`KVM_HC_KICK_CPU`	5	Wake a vCPU from HLT; `a1` = target APIC ID
`KVM_HC_SEND_IPI`	10	Multicast IPI; `a0` / `a1` = 128-bit APIC ID bitmap, `a2` = lowest APIC ID, `a3` = ICR value; up to 128 destinations per call in 64-bit mode
`KVM_HC_SCHED_YIELD`	11	Yield to scheduler when IPI target is preempted; `a0` = destination APIC ID

Sending a TLB shootdown IPI on a large guest with many vCPUs would otherwise require one APIC ICR write per destination, each of which may exit. The hypercall encodes 128 destinations in two registers, collapsing the loop to a single VM exit.

The ARM GIC (VGIC)

Arm's interrupt controller architecture is the Generic Interrupt Controller, or GIC. KVM's virtual GIC implementation is the VGIC. Interrupt IDs are organized in four ranges:

Range	IDs	Type
SGI (Software Generated)	0–15	Per-vCPU, used for IPIs
PPI (Private Peripheral)	16–31	Per-vCPU, used for timers
SPI (Shared Peripheral)	32–1019	Shared across all vCPUs
LPI (Locality-specific Peripheral)	8192+	Shared; GICv3 only

The kernel VGIC allows 64–1024 IRQs in steps of 32, configured via `KVM_DEV_ARM_VGIC_GRP_NR_IRQS`. Total IRQ count in Firecracker is `GSI_LEGACY_NUM (32) + SPI count`.

GICv2

GICv2 uses MMIO exclusively. The device type for `KVM_CREATE_DEVICE` is `KVM_DEV_TYPE_ARM_VGIC_V2`. Two MMIO regions must be placed via `KVM_DEV_ARM_VGIC_GRP_ADDR`:

Attribute	Alignment	Region size
`KVM_VGIC_V2_ADDR_TYPE_DIST`	4 KiB	4 KiB
`KVM_VGIC_V2_ADDR_TYPE_CPU`	4 KiB	8 KiB

The distributor holds global state; the CPU interface, one per vCPU, is the per-CPU window through which a running vCPU reads pending priority and signals EOI.

GICv3

GICv3 replaces the per-CPU MMIO CPU interface with system registers — the ICC and ICH register families, accessed via MSR/MRS rather than memory loads and stores. This makes EOI and priority reads faster on real hardware, and KVM emulates the same path via `KVM_DEV_ARM_VGIC_GRP_CPU_SYSREGS`. Device type: `KVM_DEV_TYPE_ARM_VGIC_V3`.

The memory layout changes substantially. Where GICv2 has one CPU interface region for all vCPUs, GICv3 introduces a **redistributor** — a 128 KiB (`KVM_VGIC_V3_REDIST_SIZE = 0x20000`) per-vCPU MMIO region that holds per-CPU state and LPI pending bits. The distributor grows to 64 KiB (`KVM_VGIC_V3_DIST_SIZE = 0x10000`):

Attribute	Alignment	Region size
`KVM_VGIC_V3_ADDR_TYPE_DIST`	64 KiB	64 KiB
`KVM_VGIC_V3_ADDR_TYPE_REDIST`	64 KiB	128 KiB per vCPU

The redistributor is selected by MPIDR (the Multiprocessor Affinity Register), so the same redistributor address space can serve multiple vCPUs if they have distinct MPIDR values.

The Interrupt Translation Service

LPIs are Locality-specific Peripheral Interrupts — GICv3's mechanism for MSI delivery. A device writes to an Interrupt Translation Table (ITT) rather than asserting a wire; the Interrupt Translation Service (ITS) translates the write into an LPI. KVM exposes this via `KVM_DEV_TYPE_ARM_VGIC_ITS` with a 128 KiB MMIO region (`GIC_V3_ITS_SIZE = 0x20000` in Firecracker), placed at a 64 KiB-aligned address via `KVM_VGIC_ITS_ADDR_TYPE`.

The ITS maintains three tables: a Device Table mapping DeviceID to an Interrupt Translation Table, an Interrupt Translation Entry table mapping EventID to a physical LPI number, and a Collection Table mapping collection IDs to redistributors. Key control registers in the ITS MMIO space: `GITS_CBASER` (command queue base address), `GITS_CWRITER` and `GITS_CREADR` (command queue write and read pointers), and `GITS_CTLR` (enable).

Lifecycle Constraints

The VGIC imposes strict ordering on its initialization sequence. `KVM_DEV_ARM_VGIC_CTRL_INIT` must be called after all vCPUs are created, because the redistributor count is determined by the vCPU count. LPI pending state — which LPIs were pending at snapshot time — must be flushed to guest RAM before a snapshot with `KVM_DEV_ARM_VGIC_SAVE_PENDING_TABLES`, and reloaded at restore time.

Firecracker places the GICv3 regions at fixed offsets below `MMIO32_MEM_START`: the distributor at `MMIO32_MEM_START - 0x10000`, the redistributors at `dist_addr - (vcpu_count * 0x20000)`, and the ITS at `redist_addr - 0x20000`.

Why Clocks Are Hard Under Virtualization

Interrupt delivery can be made fast with the mechanisms above. Timekeeping is harder because the inaccuracy accumulates invisibly and manifests far from its cause.

The time-stamp counter — `RDTSC` on x86 — is the cheapest way to measure elapsed time on a running CPU. The instruction is not serializing: out-of-order execution can produce a later `RDTSC` result that is numerically less than an earlier one on a different CPU. On multi-socket systems the TSC oscillators are independent crystals that drift due to temperature and electrical variation, so `RDTSC` on CPU 0 and `RDTSC` on CPU 1 may disagree. The Linux kernel documentation is explicit: "do not trust the TSCs to remain synchronized on NUMA or multiple socket systems."

The TSC has also historically changed rate with CPU power states. `X86_FEATURE_CONSTANT_TSC` (CPUID.80000007H:EDX[8], the "invariant TSC" bit) guarantees the TSC runs at a constant rate across all ACPI P-, C-, and T-states. Without it, any transition to a lower-frequency P-state or a shallow C-state corrupts time measurements. `X86_FEATURE_NONSTOP_TSC` guarantees the counter does not halt in C-states. On a virtualized system the guest has no control over which C-states the host enters, so it cannot rely on either property being present unless the hypervisor advertises them.

Live migration to a different host is the hardest case. The destination host's TSC may run at a slightly higher or lower frequency than the source. A faster destination TSC cannot be slowed; the hypervisor must apply an offset and a scaling factor to make the guest's visible TSC advance at the same rate.

Legacy timekeeping via the PIT (Programmable Interval Timer, I/O ports `0x40` – `0x43`, base frequency 1.193182 MHz) or the RTC (32.768 kHz crystal) relies on interrupt delivery rates that the hypervisor cannot always guarantee. When a host CPU is overloaded, timer interrupts are late; the guest's time drifts forward relative to wall clock.

Both VMX and SVM virtualize the TSC with an offset field — `TSC_OFFSET` in the VMCS; the equivalent field in the VMCB on AMD — so the guest reads `host_TSC + offset`. Both also support TSC scaling: VMX provides a 64-bit `TSC_MULTIPLIER` field in the VMCS encoded as a 48.16 fixed-point number; AMD SVM provides the `TSC_RATIO` MSR at `0xC0010104`. KVM programs these fields via the `KVM_SET_TSC_KHZ` ioctl (gated by `KVM_CAP_TSC_CONTROL` for per-vCPU control, `KVM_CAP_VM_TSC_CONTROL` for a VM-wide default applied to subsequently created vCPUs). `KVM_GET_TSC_KHZ` returns a negative error if the host TSC is unstable.

The `IA32_TSC_ADJUST` MSR (`0x3B`) provides a per-logical-processor offset added to the hardware TSC on every guest read. Its reset value is 0. Linux guests use `IA32_TSC_ADJUST` for TSC synchronization across CPUs rather than writing `IA32_TSC` directly, which would disrupt offset-based invariants. KVM tracks `IA32_TSC_ADJUST` separately from the VMCS `TSC_OFFSET`.

kvmclock: The Paravirtual Clock

The structural solution to these problems is to remove the guest's dependence on hardware clocks entirely and give it a shared-memory ABI the hypervisor updates directly. That ABI is **kvmclock**, also known as **pvclock** from the name of the data structures it uses.

The guest detects KVM with `CPUID` leaf `0x40000000`; the signature at `EBX:ECX:EDX` spells `"KVMKVMKVM\0\0\0"`. Feature bits are at leaf `0x40000001` EAX:

Bit	Constant	Meaning
0	`KVM_FEATURE_CLOCKSOURCE`	kvmclock available at deprecated MSRs `0x11` / `0x12`
3	`KVM_FEATURE_CLOCKSOURCE2`	kvmclock at canonical MSRs `0x4b564d00` / `0x4b564d01`
5	`KVM_FEATURE_STEAL_TIME`	steal time at MSR `0x4b564d03`
24	`KVM_FEATURE_CLOCKSOURCE_STABLE_BIT`	host guarantees no per-CPU warp; enables vDSO fast path

The detection algorithm: check `kvm_para_available()`, then read `cpuid_eax(0x40000001)`. If bit 3 is set, use the canonical MSRs `MSR_KVM_SYSTEM_TIME_NEW` (`0x4b564d01`) and `MSR_KVM_WALL_CLOCK_NEW` (`0x4b564d00`); if only bit 0 is set, use the deprecated pair `0x12` / `0x11`.

The Wall Clock

The guest writes a 4-byte-aligned guest physical address to `MSR_KVM_WALL_CLOCK_NEW` (`0x4b564d00`). The hypervisor fills the structure at that address:

```
struct pvclock_wall_clock {
    u32 version;  /* seqlock */
    u32 sec;      /* seconds since Unix epoch at guest boot */
    u32 nsec;
} __attribute__((__packed__));
```

This MSR is global — not per-vCPU — and records the wall time at the moment the MSR was written. To compute current wall time, the guest adds `pvclock_wall_clock.sec/nsec` to the elapsed system time obtained from `MSR_KVM_SYSTEM_TIME_NEW`.

The System Time Clock

`MSR_KVM_SYSTEM_TIME_NEW` (`0x4b564d01`) is per-vCPU. The guest writes a 4-byte-aligned guest physical address with bit 0 as the enable bit. The hypervisor fills and periodically updates the structure at that address:

```
struct pvclock_vcpu_time_info {
    u32 version;           /* seqlock; odd = update in progress */
    u32 pad0;
    u64 tsc_timestamp;     /* host TSC at last update */
    u64 system_time;       /* host monotonic ns at last update */
    u32 tsc_to_system_mul; /* fixed-point multiplier */
    s8  tsc_shift;         /* shift before multiply: positive=left, negative=right */
    u8  flags;
    u8  pad[2];
} __attribute__((packed));  /* 32 bytes total */
```

The comment in `arch/x86/include/asm/pvclock-abi.h` is unambiguous: "these structs MUST NOT be changed" — they are stable ABI shared between KVM and Xen guests.

Reading the Clock

The conversion from TSC ticks to nanoseconds uses the multiplier and shift:

```
delta = current_tsc - tsc_timestamp
if (tsc_shift >= 0): delta <<= tsc_shift
else:                delta >>= -tsc_shift
time_ns = ((delta * tsc_to_system_mul) >> 32) + system_time
```

The `version` field is a seqlock: read it before and after capturing the time fields; if either read is odd or the two values differ, the hypervisor updated the structure mid-read and the guest must retry. The seqlock protocol is what makes the update safe without a kernel lock in the guest read path.

Flags

Bit	Value	Meaning
0	1	Timestamps across CPUs are guaranteed monotonic; no global atomic needed per read
1	2	Guest vCPU was paused by the host; clear this flag and touch watchdogs

Bit 0 is set when the host advertises `KVM_FEATURE_CLOCKSOURCE_STABLE_BIT` (bit 24 in `CPUID.0x40000001`). Without it, `pvclock.c` enforces global monotonicity by updating a `last_value` counter via atomic compare-and-swap on every read — because without the host's guarantee, an unlucky migration could move the guest to a CPU with a slightly earlier TSC value, causing time to appear to go backward. With bit 0 set, the raw computed value is returned directly with no global serialization, enabling per-CPU vDSO reads.

kvmclock as a Linux Clocksource

`kvmclock_init()` in `arch/x86/kernel/kvmclock.c` registers `kvm_clock` with `clocksource_register_hz(&kvm_clock, NSEC_PER_SEC)`. The default clocksource rating is **400**, which wins over the HPET (rating 250) and the ACPI PM timer (rating 200). When the host exposes both `X86_FEATURE_CONSTANT_TSC` and `X86_FEATURE_NONSTOP_TSC` and `!check_tsc_unstable()`, the rating is reduced to **299** so that the native TSC clocksource — which is cheaper, requiring no shared-memory read — can win instead.

When `flags` bit 0 is set, `kvmclock` also calls `kvm_sched_clock_init()` to register the scheduler clock and exposes the fast path via vDSO, so that `clock_gettime(CLOCK_MONOTONIC, ...)` is serviced by a user-space shared-library read rather than a system call.

Per-vCPU `pvclock_vcpu_time_info` structures: the boot CPU uses a static array; hotplugged CPUs use dynamic allocation in `kvmclock_setup_percpu()`. TSC frequency is retrieved from these structures via `kvm_get_tsc_khz()`.

Steal Time

The paravirtual clock tells the guest how much time has elapsed from the host's perspective. Steal time tells it how much of that elapsed time the guest vCPU was actually scheduled — the complement of what the host scheduler gave to other workloads.

The guest writes a **64-byte-aligned** guest physical address (stricter than the 4-byte alignment required by the clock MSRs) with bit 0 as the enable bit to `MSR_KVM_STEAL_TIME` (`0x4b564d03`). The structure must be zero-initialized before the MSR write. The hypervisor fills:

```
struct kvm_steal_time {
    __u64 steal;      /* ns vCPU was not scheduled (excludes idle time) */
    __u32 version;    /* seqlock; even/odd protocol */
    __u32 flags;      /* currently always 0 */
    __u8  preempted;  /* nonzero = vCPU currently descheduled */
    __u8  u8_pad[3];
    __u32 pad[11];
};
```

The `steal` field counts only involuntary non-run time — host-scheduler preemption — not idle time. A guest CPU consuming 100% of its allowed time shows zero steal; a vCPU that the host is not scheduling shows increasing steal. The `preempted` field is a hint the guest can use to avoid spinning on spinlocks when it knows the vCPU holding the lock has been descheduled.

VM-Level Clock Ioctls

Two VM-level ioctls expose the kvmclock value to the VMM for snapshot and restore purposes, gated by `KVM_CAP_ADJUST_CLOCK`:

```
struct kvm_clock_data {
    __u64 clock;      /* kvmclock nanosecond value */
    __u32 flags;
    __u32 pad0;
    __u64 realtime;   /* host CLOCK_REALTIME at snapshot (if KVM_CLOCK_REALTIME set) */
    __u64 host_tsc;   /* host TSC at snapshot (if KVM_CLOCK_HOST_TSC set) */
    __u32 pad[4];
};
```

`KVM_GET_CLOCK` reads the current kvmclock value; `KVM_SET_CLOCK` restores it. Flags in the structure:

Constant	Value	Meaning
`KVM_CLOCK_TSC_STABLE`	2	`clock` is consistent across all vCPUs
`KVM_CLOCK_REALTIME`	`1 << 2`	`realtime` field is valid
`KVM_CLOCK_HOST_TSC`	`1 << 3`	`host_tsc` field is valid

`KVM_KVMCLOCK_CTRL` is a vCPU ioctl that sets a flag in the KVM vCPU state indicating the vCPU was paused by host userspace. The guest checks this flag and skips soft-lockup watchdog triggers for the duration of the pause — preventing false lockup reports when the VMM deliberately pauses vCPUs for snapshotting.

Clocks And Snapshots

The interaction between kvmclock and snapshot/restore is subtle enough to have produced multiple Firecracker bugs over several releases. Each fix is instructive.

Firecracker v1.8.0 addressed `MSR_IA32_TSC_DEADLINE` behavior at restore. If the guest sets a TSC deadline timer and then the vCPU is snapshotted, the saved `MSR_IA32_TSC_DEADLINE` may be 0 at snapshot time (because the timer already fired). Restoring a 0 deadline means the timer never fires again, stalling any guest that relies on TSC deadline interrupts. Firecracker v1.8.0 detects this condition and substitutes the current `MSR_IA32_TSC` value. It also changed restore ordering to apply `MSR_IA32_TSC_DEADLINE` after `MSR_IA32_TSC`, because KVM uses the guest TSC value when computing the deadline's meaning.

Firecracker v1.10.0 added a `KVM_KVMCLOCK_CTRL` call after pausing vCPUs on x86_64. Previously, paused vCPUs could trigger the guest's soft-lockup watchdog because the guest did not know a pause was intentional. Non-fatal failures from this ioctl increment the `vcpu.kvmclock_ctrl_fails` metric rather than aborting the pause.

Firecracker v1.16.0 fixed two separate kvmclock problems. First, guests using `kvm-clock` and running on host Linux 5.16 or later experienced the monotonic clock jumping forward by the wall-clock time elapsed since snapshot creation. The root cause: `KVM_SET_CLOCK` was not being called at restore, so the

guest's internal monotonic time and the host's wall clock could diverge. The `LoadSnapshot` API now accepts an optional `clock_realtime: true` flag that opts into calling `KVM_SET_CLOCK` with `KVM_CLOCK_REALTIME` at restore time. Without the flag, the guest monotonic clock resumes from the snapshot timestamp — a deliberate choice for workloads that want time to appear frozen across the pause.

The same v1.16.0 release fixed snapshot serialization: the snapshot previously covered only a small subset of KVM custom MSRs, missing entries such as `MSR_KVM_ASYNC_PF_INT` (`0x4b564d06`). The fix extended coverage to the full KVM custom MSR range `0x4b564d00` – `0x4b564dff` .

Firecracker exposes `kvm-clock` and `tsc` as clocksources on x86_64, and `arch_sys_counter` on aarch64. After snapshot restore, the guest wall clock continues from the moment of snapshot creation. For workloads that need accurate wall time, the recommended remediation is to update the clock post-restore via NTP or a guest agent — the VMM cannot safely correct it without risking time monotonicity violations visible to running guest processes.

Sources And Further Reading

- KVM API documentation (canonical): https://www.kernel.org/doc/html/latest/virt/kvm/api.html
- KVM x86 MSR documentation: https://www.kernel.org/doc/html/latest/virt/kvm/x86/msr.html
- KVM x86 hypercalls: https://www.kernel.org/doc/html/latest/virt/kvm/x86/hypercalls.html
- KVM x86 timekeeping: https://docs.kernel.org/virt/kvm/x86/timekeeping.html
- KVM x86 CPUID: https://docs.kernel.org/virt/kvm/x86/cpuid.html
- ARM VGICv2: https://docs.kernel.org/virt/kvm/devices/arm-vgic.html
- ARM VGICv3: https://docs.kernel.org/virt/kvm/devices/arm-vgic-v3.html
- ARM ITS: https://docs.kernel.org/virt/kvm/devices/arm-vgic-its.html
- `include/uapi/linux/kvm.h` : https://github.com/torvalds/linux/blob/master/include/uapi/linux/kvm.h
- `arch/x86/include/uapi/asm/kvm.h` : https://github.com/torvalds/linux/blob/master/arch/x86/include/uapi/asm/kvm.h
- `arch/x86/kvm/lapic.c` : https://github.com/torvalds/linux/blob/master/arch/x86/kvm/lapic.c
- `arch/x86/kvm/ioapic.h` : https://github.com/torvalds/linux/blob/master/arch/x86/kvm/ioapic.h
- `virt/kvm/eventfd.c` : https://github.com/torvalds/linux/blob/master/virt/kvm/eventfd.c
- `arch/x86/kernel/kvmclock.c` : https://github.com/torvalds/linux/blob/master/arch/x86/kernel/kvmclock.c
- `arch/x86/kernel/pvclock.c` : https://github.com/torvalds/linux/blob/master/arch/x86/kernel/pvclock.c
- `arch/x86/include/asm/pvclock-abi.h` : https://github.com/torvalds/linux/blob/master/arch/x86/include/asm/pvclock-abi.h

- `arch/x86/include/uapi/asm/kvm_para.h`: https://github.com/torvalds/linux/blob/master/arch/x86/include/uapi/asm/kvm_para.h
- Intel VT-d Posted Interrupts (Linux Foundation slides): https://events.static.linuxfound.org/sites/events/files/slides/VT-d%20Posted%20Interrupts-final%20.pdf
- Intel SDM Vol 3, posted-interrupt processing: https://xem.github.io/minix86/manual/intel-x86-and-64-manual-vol3/o_fe12b1e2a880e0ce-1179.html
- `eventfd(2)` man page: https://man7.org/linux/man-pages/man2/eventfd.2.html
- Original ioeventfd patch (LKML, August 2009): https://lkml.iu.edu/hypermail/linux/kernel/0908.2/03024.html
- LWN: KVM — add support for irqfd (2009): https://lwn.net/Articles/332924/
- OASIS virtio v1.2 CS 01: https://docs.oasis-open.org/virtio/virtio/v1.2/cs01/virtio-v1.2-cs01.html
- Firecracker `src/vmm/src/arch/x86_64/interrupts.rs`: https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/arch/x86_64/interrupts.rs
- Firecracker `src/vmm/src/device_manager/mmio.rs`: https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/device_manager/mmio.rs
- Firecracker `src/vmm/src/devices/virtio/mod.rs`: https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/devices/virtio/mod.rs
- Firecracker `src/vmm/src/vstate/vm.rs`: https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/vstate/vm.rs
- Firecracker `src/vmm/src/arch/aarch64/gic/gicv3/mod.rs`: https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/arch/aarch64/gic/gicv3/mod.rs
- Firecracker CHANGELOG: https://github.com/firecracker-microvm/firecracker/blob/main/CHANGELOG.md
- Firecracker snapshot support documentation: https://github.com/firecracker-microvm/firecracker/blob/main/docs/snapshotting/snapshot-support.md
- rust-vmm kvm-ioctls crate: https://docs.rs/kvm-ioctls/latest/kvm_ioctls/

Chapter 8: VM Exits Up Close

Every device access a guest makes — reading a byte from a UART register, writing a descriptor to a virtio queue notifier, asking `CPUID` what the CPU is — requires the hardware to stop the guest cold and hand control to the VMM. This is the VM exit, and it is both the mechanism that makes the whole architecture possible and the thing that every VMM spends the most effort avoiding.

What The Hardware Does

On Intel, the exit transfers the CPU from VMX non-root operation to VMX root operation. At the hardware level, this means the processor writes the cause to a 32-bit field in the VMCS called `VM_EXIT_REASON`. Bits 15:0 carry the **basic exit reason** — a numeric code identifying what caused the exit. Bit 31 is set only on VM-entry failures, not on exits from a running guest. Bits 28 and 29 carry special meanings for the Monitor Trap Flag and exits from VMX-root mode; they are zero in the common case.

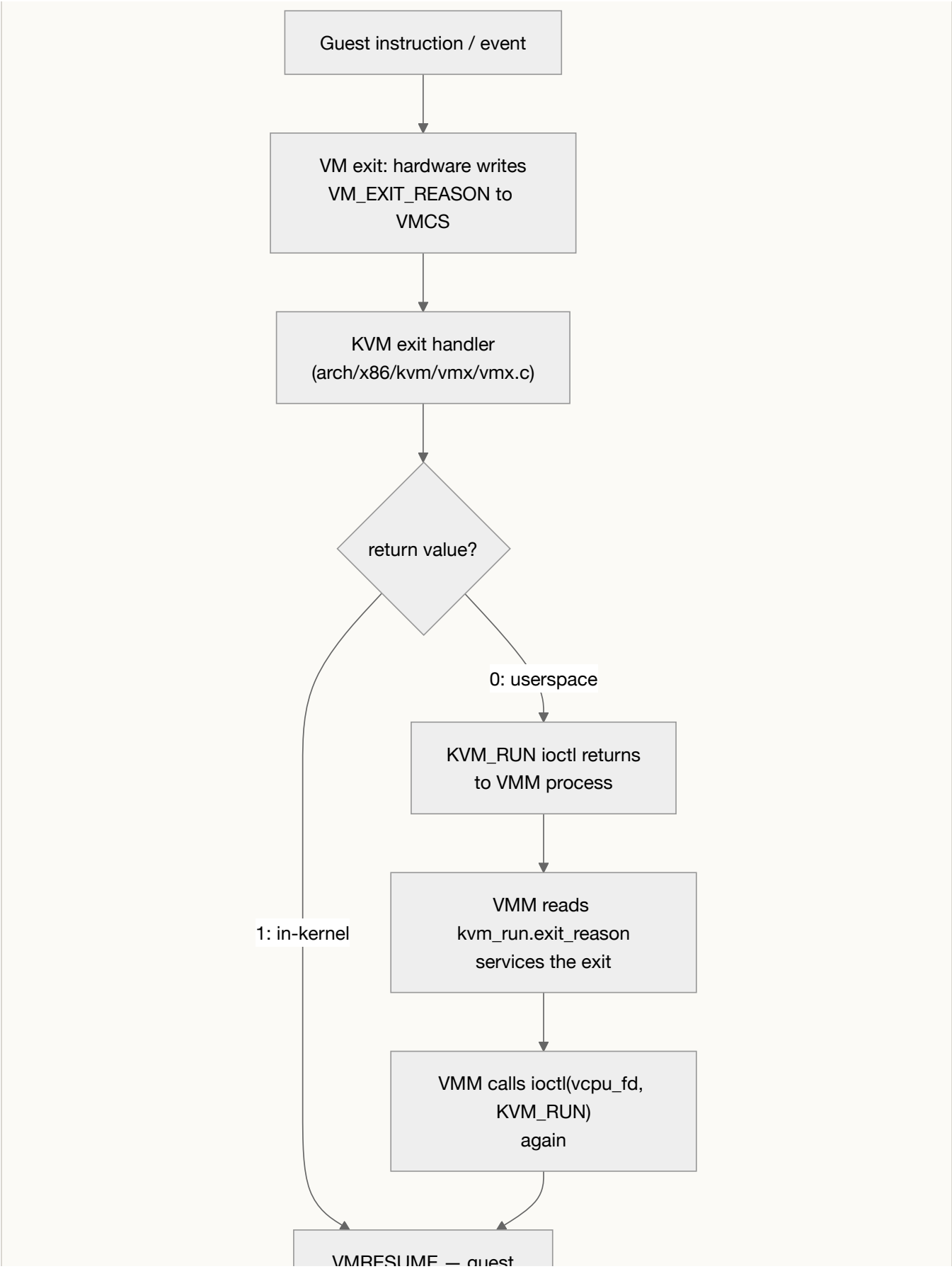
The VMCS `VM_EXIT_INSTRUCTION_LEN` field holds the byte length of the instruction that caused the exit — 2 for `CPUID`, for instance. The `EXIT_QUALIFICATION` field carries per-reason detail. For a PIO exit, `EXIT_QUALIFICATION` encodes the port number, access size, direction, and whether it was a string instruction (`INS` / `OUTS`). For an EPT violation, it encodes the access type. This detail is what KVM reads in its exit handler before deciding what to do.

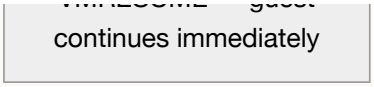
On AMD, the equivalent structure is the VMCB, and the exit writes a 64-bit exit code to the `EXITCODE` field, with additional context in `EXITINFO1` and `EXITINFO2`. The numeric values differ — PIO exits are `SVM_EXIT_IOIO = 0x07b`, CPUID is `SVM_EXIT_CPUID = 0x072`, MSR access is `SVM_EXIT_MSR = 0x07c`, triple fault is `SVM_EXIT_SHUTDOWN = 0x07f` — but the conceptual roles are identical. For an IOIO exit, `EXITINFO1` holds a bitmask: bit 0 is direction (0=OUT, 1=IN), bit 2 is a string instruction, bit 3 is a REP prefix, bits 6:4 encode the access size, and bits 31:16 hold the port number. KVM's `io_interception()` handler in `arch/x86/kvm/svm/svm.c` reads these masks and populates the same generic `kvm_vcpu_io` structure that the VMX path uses, so both architectures surface to userspace through an identical `kvm_run.io` struct.

A detail that surprises people who expect the hardware to be comprehensive: neither VMX nor SVM saves general-purpose registers automatically on exit. The hypervisor exit stub must save them before using them. VMX does save CR0, CR3, CR4, RSP, RIP, RFLAGS, segment descriptors, GDTR, IDTR, TR, and a configurable set of MSRs — but RAX through R15 are the VMM's responsibility. The host register state is restored from the VMCS host-state area, not from general-purpose register saves.

How KVM Routes Exits

KVM's `kvm_arch_vcpu_ioctl_run()` in `arch/x86/kvm/x86.c` calls `vcpu_enter_guest()`, which issues either `VMLAUNCH` or `VMRESUME`. When the guest exits, KVM reads the exit reason and dispatches to a per-reason handler. The return convention of that handler determines what happens next: a return value of `1` means KVM handled it in-kernel and re-enters the guest immediately; a return value of `0` means KVM fills `kvm_run.exit_reason` and returns from the `KVM_RUN` ioctl to userspace. The VMM process never wakes up for a return-1 exit.


Guest instruction / event
VM exit: hardware writes VM_EXIT_REASON to VMCS
KVM exit handler (arch/x86/kvm/vmx/vmx.c)
return value?
0: userspace
KVM_RUN ioctl returns to VMM process
VMM reads kvm_run.exit_reason services the exit
1: in-kernel
VMM calls ioctl(vcpu_fd, KVM_RUN) again
VMRESUME — guest



Several exit categories never reach userspace. **EPT violations on RAM** — a guest touching a mapped guest-physical address whose EPT entry does not yet exist — are resolved by `kvm_mmu_page_fault()`, which installs the mapping and resumes. The VMM never sees these; they are the hardware page-fault mechanism working normally.

`CPUID` causes an unconditional VMX exit (basic exit reason 10). KVM handles it entirely in-kernel by consulting the per-vCPU table set by `KVM_SET_CPUID2`. There is no `KVM_EXIT_CPUID` in the KVM uAPI.

MSR accesses are governed by the MSR bitmap — a 4 KB VMCS field covering MSR addresses `0x0` – `0x1fff` and `0xc0000000` – `0xc0001fff`. If an MSR's bit is clear in the bitmap, the guest reads or writes it without a VM exit at all. For MSRs outside that range that do exit, KVM's `kvm_emulate_rdmsr()` and `kvm_emulate_wrmsr()` handle the ones KVM owns in-kernel: `IA32_TSC`, `IA32_APIC_BASE`, the KVM paravirt MSR range `0x4b564d00` – `0x4b564d08`, and many others.

`HLT`, when the "HLT exiting" bit (bit 7 of the processor-based VM-execution controls) is set, sends the vCPU thread to `kvm_vcpu_block()` in `virt/kvm/kvm_main.c`. With halt polling active, the thread spins for up to `halt_poll_ns` nanoseconds checking for a pending wakeup; an interrupt arriving in that window resumes the guest with no scheduler round-trip.

In-kernel irqchip accesses — when `KVM_CREATE_IRQCHIP` and `KVM_CREATE_PIT2` are used — route APIC-register accesses and interrupt-controller MMIO through the KVM virtual device layer via `kvm_io_bus`, never surfacing to userspace.

The exits that do reach userspace are defined by constants in `include/uapi/linux/kvm.h`, part of the stable KVM ABI. The x86-relevant subset:

Value	Constant	When it fires
0	`KVM_EXIT_UNKNOWN`	Hardware exit reason KVM did not recognize
2	`KVM_EXIT_IO`	PIO `IN` / `OUT` to a port with no in-kernel owner
5	`KVM_EXIT_HLT`	Guest `HLT` when not handled in-kernel
6	`KVM_EXIT_MMIO`	MMIO to a GPA not backed by RAM or an in-kernel device
8	`KVM_EXIT_SHUTDOWN`	Triple fault
9	`KVM_EXIT_FAIL_ENTRY`	Hardware refused VM entry
17	`KVM_EXIT_INTERNAL_ERROR`	KVM subsystem error
29	`KVM_EXIT_X86_RDMSR`	RDMSR delegated to userspace (requires `KVM_CAP_X86_USER_SPACE_MSR`)
30	`KVM_EXIT_X86_WRMSR`	WRMSR delegated to userspace (requires `KVM_CAP_X86_USER_SPACE_MSR`)

The kvm_run Structure

The VMM maps the `struct kvm_run` page once, before the first `KVM_RUN`:

```
int mmap_size = ioctl(kvm_fd, KVM_GET_VCPU_MMAP_SIZE, 0);
struct kvm_run *run = mmap(NULL, mmap_size, PROT_READ|PROT_WRITE,
                           MAP_SHARED, vcpu_fd, 0);
```

This page persists across `KVM_RUN` calls. The VMM reads it after each ioctl returns. The abbreviated layout, from `include/uapi/linux/kvm.h`:

```
struct kvm_run {
    /* VMM writes before KVM_RUN */
    __u8  request_interrupt_window;  /* exit when guest IF opens */
    __u8  immediate_exit;            /* force exit after one instruction */
    __u8  padding1[6];

    /* KVM writes after exit */
    __u32 exit_reason;               /* KVM_EXIT_* */
    __u8  ready_for_interrupt_injection;
    __u8  if_flag;
    __u16 flags;

    /* in/out */
    __u64 cr8;
    __u64 apic_base;

    union {
        struct {                     /* KVM_EXIT_IO */
            __u8  direction;         /* 0 = IN, 1 = OUT */
            __u8  size;              /* 1, 2, or 4 bytes */
            __u16 port;
            __u32 count;             /* repetition count for REP INS/OUTS */
            __u64 data_offset;       /* offset from start of kvm_run to data */
        } io;

        struct {                     /* KVM_EXIT_MMIO */
            __u64 phys_addr;         /* guest physical address */
            __u8  data[8];           /* fill on read, read on write */
            __u32 len;
            __u8  is_write;
        } mmio;

        struct {                     /* KVM_EXIT_X86_RDMSR / KVM_EXIT_X86_WRMSR */
            __u8  error;             /* out: 0 = ok, 1 = inject #GP */
            __u8  pad[7];
            __u32 reason;            /* KVM_MSR_EXIT_REASON_* bitmask */
            __u32 index;             /* MSR address (RCX) */
            __u64 data;              /* RDMSR fill / WRMSR value */
        } msr;

        struct {                     /* KVM_EXIT_FAIL_ENTRY */
            __u64 hardware_entry_failure_reason;
            __u32 cpu;
        } fail_entry;

        struct {                     /* KVM_EXIT_INTERNAL_ERROR */
            __u32 suberror;
            __u32 ndata;
            __u64 data[16];
        } internal;
```

```
        /* KVM_EXIT_HLT: exit_reason alone is the signal; no union member */
        /* KVM_EXIT_SHUTDOWN: no union member */

        char padding[256];
    };

    __u64 kvm_valid_regs;
    __u64 kvm_dirty_regs;
};
```

One detail that trips people: the `KVM_EXIT_IO` data is not stored inside the struct. The `io.data_offset` field is a byte offset from the start of the `kvm_run` page. The VMM accesses it as `(char *)run + run->io.data_offset`. The separation exists because REP string I/O (`INS`, `OUTS`) can move more data than fits in eight bytes, and inline storage would overflow other struct fields.

Servicing Each Exit

PIO: KVM_EXIT_IO

When a guest executes an `IN` or `OUT` instruction to a port that is not in-kernel owned, KVM fills `run->io` and returns. The VMM reads `run->io.port`, `run->io.direction`, `run->io.size`, and `run->io.count`. For an `OUT` (guest write, `direction == 1`), the data is already in the buffer at `data_offset`. For an `IN` (guest read, `direction == 0`), the VMM must fill the buffer before returning to `KVM_RUN`. The canonical pattern, from the LWN KVM tutorial:

```
case KVM_EXIT_IO:
    if (run->io.direction == KVM_EXIT_IO_OUT && run->io.port == 0x3f8)
        putchar(*((char *)run + run->io.data_offset));
    break;
```

Port `0x3f8` is the first COM port, the base address of the 16550A UART — where any Linux guest sends its early boot messages. That one line is the complete device model for a serial console in a toy VMM.

In Firecracker, the dispatch is more structured. In `src/vmm/src/arch/x86_64/vcpu.rs`, `VcpuExit::IoIn(addr, data)` and `VcpuExit::IoOut(addr, data)` are delivered to `pio_bus.read()` and `pio_bus.write()`. The `pio_bus` is a sorted map of address ranges to registered device handlers. If the port has no registered handler, the read data is zero-filled with a `warn!` log and the exit returns `Handled` — the guest sees zeros rather than a fault.

MMIO: KVM_EXIT_MMIO

MMIO exits arise when the guest accesses a guest-physical address that is not backed by a RAM slot and has no in-kernel device handler. On Intel, the EPT records the access as a violation (basic exit reason 48); on AMD, a nested page fault (`SVM_EXIT_NPF = 0x400`) on an address without an NPT entry serves the

same role. KVM identifies both as MMIO from the absent mapping and surfaces them as `KVM_EXIT_MMIO`.

The VMM reads `run->mmio.phys_addr`, `run->mmio.len`, and `run->mmio.is_write`. For a guest write, `run->mmio.data` already contains the value. For a guest read, the VMM fills `run->mmio.data` before returning.

Firecracker handles this in `src/vmm/src/vstate/vcpu.rs`:

```
VcpuExit::MmioRead(addr, data) => {
    data.fill(0);
    if let Some(mmio_bus) = &peripherals.mmio_bus {
        mmio_bus.read(addr, data)?;
        METRICS.vcpu.exit_mmio_read.inc();
    }
    Ok(VcpuEmulation::Handled)
}
VcpuExit::MmioWrite(addr, data) => {
    if let Some(mmio_bus) = &peripherals.mmio_bus {
        mmio_bus.write(addr, data)?;
        METRICS.vcpu.exit_mmio_write.inc();
    }
    Ok(VcpuEmulation::Handled)
}
```

Firecracker maintains two separate `Bus` structs: `mmio_bus` for virtio device config-space accesses and `pio_bus` for legacy PIO devices (the UART, the i8042). Each is a sorted map of address ranges to device handlers, and the dispatch is a binary search. Every MMIO access the guest makes to a virtio device register — DeviceID, DeviceFeatures, QueueNotify — comes through this path.

HLT: KVM_EXIT_HLT

`KVM_EXIT_HLT` (value 5) carries no data fields; the exit reason is the complete signal. Under normal KVM configuration, HLT is handled in-kernel by `kvm_vcpu_block()`, which parks the vCPU thread until a wakeup event arrives. The exit surfaces to userspace only when the in-kernel handler cannot manage it.

Firecracker treats `VcpuExit::Hlt` as an unhandled exit — it appears as `UnhandledKvmExit("Hlt")` in the generic `handle_kvm_exit()` in `src/vmm/src/vstate/vcpu.rs`. This is deliberate: Firecracker expects guests to use the ACPI power-off path (`KVM_EXIT_SYSTEM_EVENT`) for orderly shutdown, not HLT. A bare `HLT` in a Firecracker guest that somehow escapes the in-kernel handler would be treated as an error condition.

CPUID: No Userspace Exit

`CPUID` causes an unconditional VM exit on both Intel (basic exit reason 10) and AMD (`SVM_EXIT_CPUID = 0x072`), but KVM handles it entirely in-kernel. There is no `KVM_EXIT_CPUID` in the uAPI. The VMM calls `KVM_SET_CPUID2` once per vCPU before the first `KVM_RUN` to pre-load the CPUID table KVM

consults on each exit.

The KVM paravirt signature, placed at synthetic leaf `0x40000000`, reads "KVMKVMKVM\0" across EBX, ECX, EDX. Leaf `0x40000001` carries feature bits: bit 0 is `KVM_FEATURE_CLOCKSOURCE`, bit 3 is `KVM_FEATURE_CLOCKSOURCE2`, bit 5 is `KVM_FEATURE_STEAL_TIME`, bit 6 is `KVM_FEATURE_PV_EOI`, and bit 24 is `KVM_FEATURE_CLOCKSOURCE_STABLE_BIT`. A guest that finds this signature at leaf `0x40000000` knows it is running under KVM and can opt into the paravirt interfaces those bits advertise.

Firecracker does not simply pass through the host `CPUID` to the guest. It applies a `CpuidLeafModifier` template — a set of bitmask operations per leaf — and then runs a normalization pass in `cpuid/normalize.rs` that enforces correct topology in leaf `0x1` (the APIC ID and HTT bit), fills leaf `0xB` extended topology, and clears or sets specific feature bits for reproducibility across hosts. The goal is a guest that sees the same CPU regardless of which physical host it lands on — essential when snapshots may be restored on different hardware.

MSR Exits: KVM_EXIT_X86_RDMSR and KVM_EXIT_X86_WRMSR

Most MSR accesses never reach userspace. KVM's MSR bitmap suppresses exits for MSRs it handles fully — `IA32_TSC`, `IA32_APIC_BASE`, the KVM PV MSR range, and dozens of others. For MSRs outside that range that do exit, KVM normally injects `#GP` into the guest.

Delegating unknown MSR accesses to userspace requires two capabilities set explicitly: `KVM_CAP_X86_USER_SPACE_MSR` (capability 188) and optionally `KVM_CAP_X86_MSR_FILTER` (capability 189). With `KVM_CAP_X86_USER_SPACE_MSR` enabled, unknown or filtered MSRs produce `KVM_EXIT_X86_RDMSR` (29) or `KVM_EXIT_X86_WRMSR` (30). The VMM then reads `run->msr.index` (the MSR address from `RCX`), fills `run->msr.data` for a read, reads `run->msr.data` for a write, and sets `run->msr.error = 0` for success or `1` to inject `#GP` into the guest.

The `KVM_MSR_EXIT_REASON_*` flags describe why the exit occurred:

```
KVM_MSR_EXIT_REASON_INVAL   (1 << 0)  /* invalid MSR or reserved bits */
KVM_MSR_EXIT_REASON_UNKNOWN (1 << 1)  /* KVM has no handler for this MSR */
KVM_MSR_EXIT_REASON_FILTER  (1 << 2)  /* blocked by KVM_X86_SET_MSR_FILTER */
```

Firecracker does not use `KVM_CAP_X86_USER_SPACE_MSR` and does not handle `KVM_EXIT_X86_RDMSR` or `KVM_EXIT_X86_WRMSR` in its run loop. MSRs are configured at boot via `KVM_SET_MSRS` using CPU template `RegisterValueFilter` entries, and KVM handles them from there.

The KVM paravirt MSR range `0x4b564d00` – `0x4b564d08` is handled in-kernel. The addresses and their purposes:

MSR address	Purpose
`0x4b564d00`	`MSR_KVM_WALL_CLOCK_NEW` — guest wallclock GPA
`0x4b564d01`	`MSR_KVM_SYSTEM_TIME_NEW` — per-vCPU monotonic time GPA
`0x4b564d02`	`MSR_KVM_ASYNC_PF_EN` — async page fault control
`0x4b564d03`	`MSR_KVM_STEAL_TIME` — vCPU steal-time GPA
`0x4b564d04`	`MSR_KVM_EOI_EN` — PV EOI control
`0x4b564d05`	`MSR_KVM_POLL_CONTROL` — host halt-polling enable/disable
`0x4b564d06`	`MSR_KVM_ASYNC_PF_INT` — APF "page ready" interrupt vector
`0x4b564d07`	`MSR_KVM_ASYNC_PF_ACK` — APF acknowledgement
`0x4b564d08`	`MSR_KVM_MIGRATION_CONTROL` — live migration permission

The older addresses `0x11` (`MSR_KVM_WALL_CLOCK`) and `0x12` (`MSR_KVM_SYSTEM_TIME`) are deprecated; guests should use the `0x4b564d00` variants when bit 3 of leaf `0x40000001` is set.

Shutdown: KVM_EXIT_SHUTDOWN

A guest triple fault delivers `KVM_EXIT_SHUTDOWN` (value 8). No fields in the `kvm_run` union carry detail; the exit reason alone is the signal. A triple fault occurs when the CPU encounters a fault while trying to deliver another fault — typically a fault with no valid IDT handler and then another fault while processing that. The hardware has nowhere to go. On AMD, this is `SVM_EXIT_SHUTDOWN = 0x07f`.

The VMM must terminate the vCPU thread. Re-entering the guest after a shutdown exit is undefined behavior. Firecracker tears down the vCPU and reports the event. In a properly functioning guest, shutdown flows through ACPI: the guest writes a power-off request, the VMM receives `KVM_EXIT_SYSTEM_EVENT`, and the vCPU loop terminates cleanly. A triple fault is an aberration — a guest kernel crash or a bug that cascades past all fault handlers.

Internal Error: KVM_EXIT_INTERNAL_ERROR

`KVM_EXIT_INTERNAL_ERROR` (value 17) signals that KVM itself encountered a problem. The `run->internal.suberror` field carries a sub-code. With `KVM_CAP_EXIT_ON_EMULATION_FAILURE` (capability 204) enabled, instruction emulation failures produce suberror `KVM_INTERNAL_ERROR_EMULATION`; the detailed sub-struct then contains the VM exit reason, `exit_info1` and `exit_info2`, and up to 15 bytes of the failing instruction. This is the right way to handle emulation failures: the VMM can log the instruction bytes and exit cleanly rather than guessing what went wrong.

Without that capability, KVM injects `#UD` (invalid opcode) silently. This is a hard bug to diagnose — the guest crashes on a valid instruction with no indication from KVM that emulation failed.

The Canonical Run Loop

The exit dispatch structure is the same across all VMMs. Here is the loop stripped to its essentials:

```
for (;;) {
    ioctl(vcpu_fd, KVM_RUN, NULL);     /* 0xAE80 = _IO(0xAE, 0x80) */
    switch (run->exit_reason) {
    case KVM_EXIT_HLT:
        return 0;
    case KVM_EXIT_IO:
        /* direction, size, port, count: run->io.*                  */
        /* data: (char *)run + run->io.data_offset                  */
        break;
    case KVM_EXIT_MMIO:
        /* phys_addr, len, is_write: run->mmio.*                    */
        /* for reads: fill run->mmio.data before returning          */
        break;
    case KVM_EXIT_SHUTDOWN:
        abort();
    case KVM_EXIT_INTERNAL_ERROR:
        /* run->internal.suberror for detail */
        abort();
    }
}
```

The `KVM_RUN` ioctl encodes as `_IO(KVMIO, 0x80)` where `KVMIO = 0xAE`, giving the raw value `0xAE80`. It takes no parameter.

What An Exit Costs

No Intel or AMD specification publishes a cycle count for VM exit and re-entry. All available measurements are empirical, and they vary substantially across microarchitectures and workloads. VMXbench (utshina) measured roughly 330 cycles for the exit and 294 cycles for the entry on Skylake-era hardware, using a `VMCALL`-triggered exit and immediate `VMRESUME`. Measurements on Sandy Bridge-E (Core i7-3960X) with a custom KVM patch found round-trip latencies of roughly 1,600–1,700 cycles at the 95th percentile with pinned vCPUs. The ACRN project documentation characterizes ICR MSR accesses as approximately 3–4 microseconds.

The variance is not surprising when you consider what the hardware must do. The VMCS save-and-restore writes guest CR0, CR3, CR4, RSP, RIP, RFLAGS, segment descriptors, and any MSRs in the VM-exit MSR-save list, then reads the host values from the VMCS host-state area. TLB entries tagged to the VPID or EPTP may be flushed unless VPID is in use and neither CR3 nor EPTP changed. The exit returns the CPU to VMX-root code — KVM's exit handler — that likely evicted from L1 and L2 while the guest was running. That cold-cache penalty is real and workload-dependent.

The Spectre v2 mitigations added another cost: with retpolines enabled, indirect calls through KVM's `vmx_exit_handlers[]` function-pointer table each take the retpoline penalty. Andrea Arcangeli's "KVM Monolithic" restructuring (around 2019), merged by KVM maintainer Paolo Bonzini, reduced this by eliminating indirect dispatch across the `kvm.ko` / `kvm-intel.ko` module boundary, yielding double-digit percentage improvement on `CPUID`-heavy benchmarks with default mitigations enabled. The QEMU team documented this work in a 2019 blog post titled "Micro-Optimizing KVM VM-Exits."

At 300–1,700 cycles per round-trip and a guest capable of billions of instructions per second, one exit per microsecond (10^6 exits/s) consumes roughly 0.1–0.3% of host CPU just in exit-and-entry overhead. The actual cost is higher because device emulation runs in that window. A VMM that handles 100 exits per guest millisecond is spending a measurable fraction of a core just on overhead — before a single byte of device emulation executes.

Intel APICv (controlled by the "virtual-interrupt delivery" and "APIC-register virtualization" VM-execution controls, widely cited as introduced around the Ivy Bridge-EP / Haswell era) eliminates a large class of exits entirely. Historically, guest kernel LAPIC MMIO accesses and x2APIC `WRMSR` instructions were among the highest-frequency exit categories; APICv allows the processor to deliver virtual interrupts and update the virtual APIC page without exiting to VMX root mode at all. AMD AVIC provides the equivalent on AMD hardware.

Eliminating Exits: ioeventfd and irqfd

The two highest-frequency exit categories in a typical Firecracker microVM are virtio queue notifications (MMIO writes to the `QueueNotify` register at virtio MMIO offset `0x050`) and device interrupt injection. Both can be made zero-exit.

`KVM_IOEVENTFD` (capability `KVM_CAP_IOEVENTFD`) registers an eventfd file descriptor against a guest MMIO or PIO address. When the guest writes to that address, KVM signals the eventfd and resumes the guest — the `KVM_RUN` ioctl never returns to userspace for that write. The registration struct:

```
struct kvm_ioeventfd {
    __u64 datamatch;    /* optional: match only this write value */
    __u64 addr;         /* MMIO or PIO address */
    __u32 len;          /* access width */
    __s32 fd;           /* eventfd to signal */
    __u32 flags;
    /* KVM_IOEVENTFD_FLAG_PIO       -- PIO (default: MMIO) */
    /* KVM_IOEVENTFD_FLAG_DATAMATCH -- filter by datamatch value */
    /* KVM_IOEVENTFD_FLAG_DEASSIGN  -- remove this registration */
};
```

> ***Note:*** *`KVM_IOEVENTFD` and `KVM_IRQFD` are VM-level ioctls on the VM fd. They are typically called during device setup, before the guest has a chance to write to the target address; they do not require the vCPU threads to be stopped.*

Firecracker registers each virtio device's queue notify address via `KVM_IOEVENTFD` during device initialization. A guest write to `QueueNotify` signals the eventfd in-kernel — no exit ever reaches Firecracker's vCPU thread for the virtio data-plane hot path. Unregistered MMIO addresses still produce `KVM_EXIT_MMIO`.

`KVM_IRQFD` (capability `KVM_CAP_IRQFD`) registers an eventfd paired with a guest GSI routing entry. When the host signals the eventfd — a device backend completing a DMA transfer, a network packet arriving — KVM injects the corresponding virtual interrupt into the guest without any userspace round-trip.

Together, `ioeventfd` and `irqfd` make the virtio data plane a kernel-to-kernel path. The guest writes the queue notifier (in-kernel ioeventfd delivery), the device thread processes buffers, and signals the guest via irqfd injection. No userspace VMM code runs in the steady state. This is why Firecracker can sustain high-throughput network and disk I/O at low vCPU overhead: the hot path does not touch the vCPU run loop.

Halt Polling

When a guest `HLT`s, `kvm_vcpu_block()` in `virt/kvm/kvm_main.c` does not immediately yield the vCPU thread to the Linux scheduler. It spins first, for up to `halt_poll_ns` nanoseconds, checking for a pending wakeup event. If an interrupt arrives during that window — the common case in a guest waiting for a virtio response — the guest resumes without the cost of a scheduler round-trip, which itself runs several microseconds.

The polling interval adapts dynamically. On a successful poll (interrupt arrived during the window), the next window grows by a factor of `halt_poll_ns_grow` (default: 2), starting from `halt_poll_ns_grow_start` (default: 10,000 ns). On a failed poll (the vCPU thread yielded anyway), the window shrinks by `halt_poll_ns_shrink` (default: 2). The ceiling is `halt_poll_ns`. Check `/sys/module/kvm/parameters/halt_poll_ns` on the target host to see the current ceiling — the default varies across kernel versions.

The trade-off is CPU consumption on the host. A guest that halts frequently but wakes quickly pays nothing extra if the interrupt arrives within the polling window; it pays nothing if the polling window is zero. A guest that halts and stays halted for hundreds of microseconds wastes the entire polling window on the host before finally yielding. `halt_poll_ns` is a knob, not a fixed cost.

Instruction Emulation

Most VM exits are clean: a discrete instruction ran to completion, the exit reason names it, and the VMM can service the exit and resume. But some exits occur mid-instruction, or after an instruction the hardware could not complete. For these, KVM uses the software emulator in `arch/x86/kvm/emulate.c`.

The emulator:

1. Decodes the instruction at `GUEST_RIP` from guest memory.
2. Runs the emulated operation against KVM's model of guest register state.
3. Updates guest GPRs, flags, and RIP.

This is not a toy decoder. `arch/x86/kvm/emulate.c` handles the full x86 instruction encoding — REX prefixes, VEX extensions, operand sizes, memory addressing modes — because any instruction that triggered an exit might need emulation. The emulator is the code path KVM takes when EPT reports a write to a GPA that turns out to be MMIO, and the hardware's partial instruction needs completing in software.

Emulation failures surface as `KVM_EXIT_INTERNAL_ERROR` with suberror `KVM_INTERNAL_ERROR_EMULATION` when `KVM_CAP_EXIT_ON_EMULATION_FAILURE` (capability 204) is enabled. The sub-struct carries the exit reason, `exit_info1` and `exit_info2`, and up to 15 bytes of the failing instruction — enough to identify the problem. Without that capability, KVM injects `#UD` silently, and the guest crashes on a valid instruction with no diagnostic output from the VMM.

A concrete example of what goes wrong without careful emulation: a bug fixed in Linux ~5.15 (by Hou Wenlong) found that `x86_emulate_instruction()` did not check `vcpu->arch.complete_userspace_io` after RDMSR or WRMSR emulation through the software emulator path. The instruction was advanced past (RIP incremented) without the userspace exit being delivered. The guest's MSR read would silently return stale register values. The fix adds a check before advancing RIP.

Observability

The `kvm_stat` tool at `tools/kvm/kvm_stat/kvm_stat` in the kernel tree reads per-VM and per-vCPU exit counters from debugfs at `/sys/kernel/debug/kvm/` and prints a rolling summary by exit type. Running it against a live Firecracker instance shows which exit types dominate: on a boot-heavy workload, `io_instruction` and `mmio` exits; on a network-heavy workload, `external_interrupt` entries when IRQ injection is in use.

For per-exit profiling, the `kvm_exit` and `kvm_mmio` kernel tracepoints feed `perf kvm stat`, which aggregates exit counts and average latencies by exit type across all VMs on the host. A trace that shows thousands of MMIO exits per second to addresses that should be `ioeventfd`-registered suggests a device setup problem worth investigating.

Firecracker exposes its own counters. `METRICS.vcpu.exit_mmio_read`, `exit_mmio_write`, `exit_io_in`, and `exit_io_out` increment in the exit dispatch paths and are surfaced via Firecracker's metrics endpoint. These are the first numbers to check when a Firecracker guest shows unexpectedly high vCPU CPU consumption.

Chapter 9 builds from here — a minimal VMM whose run loop has exactly these case arms, each one earned.

Sources And Further Reading

- Linux KVM uAPI header (`kvm.h`): exit reason constants (`KVM_EXIT_*`), `struct kvm_run`, `struct kvm_ioeventfd`, capability numbers. https://github.com/torvalds/linux/blob/master/include/uapi/linux/kvm.h
- Linux SVM uAPI header (`svm.h`): AMD exit codes (`SVM_EXIT_IOIO = 0x07b`, `SVM_EXIT_MSR = 0x07c`, `SVM_EXIT_SHUTDOWN = 0x07f`, `SVM_EXIT_NPF = 0x400`). https://github.com/torvalds/linux/blob/master/arch/x86/include/uapi/asm/svm.h
- Linux SVM kernel header (`asm/svm.h`): IOIO bitmask definitions (`SVM_IOIO_TYPE_MASK`, `SVM_IOIO_STR_MASK`, `SVM_IOIO_SIZE_MASK`). https://github.com/torvalds/linux/blob/master/arch/x86/include/asm/svm.h
- KVM API documentation: definitive reference for all ioctls, capabilities, and `kvm_run` fields. https://docs.kernel.org/virt/kvm/api.html
- KVM halt polling documentation: `halt_poll_ns`, growth and shrink parameters, the polling algorithm. https://docs.kernel.org/virt/kvm/halt-polling.html
- KVM CPUID documentation: synthetic leaf `0x40000000` signature, `0x40000001` feature bits, `KVM_FEATURE_CLOCKSOURCE`. https://docs.kernel.org/virt/kvm/x86/cpuid.html
- KVM MSR documentation: paravirt MSR range `0x4b564d00` – `0x4b564d08`, deprecated MSRs `0x11` / `0x12`. https://docs.kernel.org/virt/kvm/x86/msr.html
- "Using the KVM API," LWN.net: the canonical C walkthrough of the KVM run loop, including the serial port `putchar` pattern. https://lwn.net/Articles/658511/
- LWN article on `KVM_CAP_X86_USER_SPACE_MSR` and the MSR userspace exit mechanism. https://lwn.net/Articles/1032708/
- Firecracker `vstate/vcpu.rs`: the exit dispatch loop, MMIO handler, HLT handling. https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/vstate/vcpu.rs
- Firecracker `arch/x86_64/vcpu.rs`: PIO dispatch via `pio_bus`. https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/arch/x86_64/vcpu.rs
- Firecracker CPU templates documentation: `CpuidLeafModifier`, `RegisterValueFilter`, template application. https://github.com/firecracker-microvm/firecracker/blob/main/docs/cpu_templates/cpu-templates.md

- Firecracker CPUID normalization documentation: leaf `0x1` topology, leaf `0xB` extended topology, normalization pass in `cpuid/normalize.rs`. https://github.com/firecracker-microvm/firecracker/blob/main/docs/cpu_templates/cpuid-normalization.md
- intel/haxm `vmx.h`: PIO exit qualification bit layout (bits 2:0 access size, bit 3 direction, bits 31:16 port number). https://github.com/intel/haxm/blob/master/core/include/vmx.h
- rust-vmm `kvm-ioctls` vCPU implementation: `KVM_RUN` encoding and vCPU exit mapping. https://github.com/rust-vmm/kvm/blob/main/kvm-ioctls/src/ioctls/vcpu.rs
- QEMU blog, "Micro-Optimizing KVM VM-Exits" (2019): Spectre v2 retpoline overhead on the exit path, KVM Monolithic restructuring, double-digit performance recovery. https://www.qemu.org/2019/11/15/micro-optimizing-kvm-vmexits/
- Intel SDM Vol. 3C, Table C-1: basic exit reason codes, `VM_EXIT_REASON` field layout. https://cdrdv2-public.intel.com/812396/326019-sdm-vol-3c.pdf
- `kvm_stat` tool: per-VM and per-vCPU exit counters from `/sys/kernel/debug/kvm/`. https://github.com/torvalds/linux/blob/master/tools/kvm/kvm_stat/kvm_stat

PART III — THE VIRTUAL MACHINE MONITOR

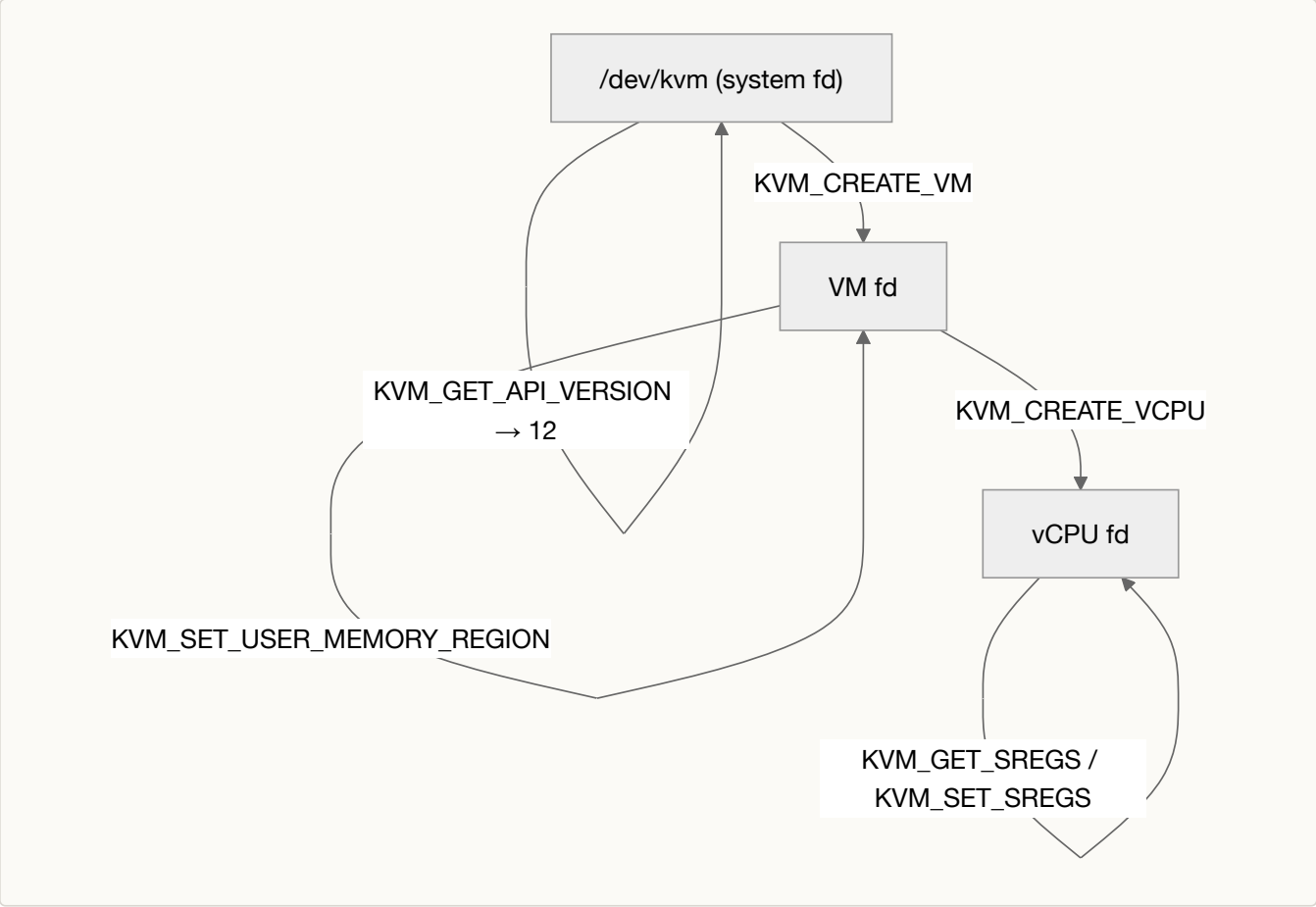
Chapter 9: Anatomy Of A VMM

Every VMM you will ever read — kvmtool, crosvm, Cloud Hypervisor, Firecracker — does the same five things before a single guest instruction retires. The wrappers differ, the languages differ, and the ambition differs, but the kernel interface is stable and narrow. Underneath each VMM is a handful of `ioctl` calls on three file descriptors, a shared page that ferries data across every VM-exit, and a thread for each guest CPU. The five jobs compose into every VMM, from the 200-line teaching example in LWN to the production runtime that boots AWS Lambda functions.

The KVM Fd Hierarchy

KVM exposes a three-level file-descriptor hierarchy. Every level is a character device opened by a different `ioctl`, and each level accepts only the `ioctl`s designed for it — send a vCPU `ioctl` to a system fd and the kernel returns `ENOTTY`.

The root is `/dev/kvm`. Opening it gives you a **system fd** whose most important job is creating VMs. Before doing that, a VMM should call `KVM_GET_API_VERSION` (`_IO(0xAE, 0x00)`) and verify the result is `12`. That number has not changed since KVM was merged; if you get anything else you are talking to a kernel too old or too exotic to trust.



`KVM_CREATE_VM` (`_IO(0xAE, 0x01)`) on the system fd yields a **VM fd** that represents the guest address space and its interrupt state. `KVM_CREATE_VCPU` (`_IO(0xAE, 0x41)`) on the VM fd yields a **vCPU fd** — one per guest core — that exposes the per-CPU register file and the interface for entering and exiting guest mode. The type byte `0xAE` is the `KVMIO` constant; it appears in every KVM `ioctl` number by convention.

The hierarchy matters because it is the security boundary. Granting a process the VM fd without the system fd limits what VMs it can create. Granting the vCPU fd without the VM fd limits what memory mappings it can see. Real VMMs like Firecracker use the `jailer` binary to enter a `chroot`, drop privileges, and pass in a pre-opened VM fd before seccomp locks down the process, so no single thread ever holds all three capabilities simultaneously after the sandbox is active.

Job 1: Allocate Guest Memory

The guest's physical address space is a lie. Every guest-physical address (GPA) is a host-virtual address (HVA) in disguise — the VMM `mmap`s an anonymous buffer, then registers a range of host virtual memory as a contiguous block of guest-physical memory. The hardware (Intel's EPT, AMD's NPT) translates GPA to HPA transparently during guest execution, using page tables the kernel maintains inside the VMCS or VMCB. The VMM never writes to those hardware tables directly; it only tells the kernel where its buffer lives.

The registration call is `KVM_SET_USER_MEMORY_REGION` (`_IOW(0xAE, 0x46, struct kvm_userspace_memory_region)`), a VM `ioctl`. The struct is compact:

```
struct kvm_userspace_memory_region {
    __u32 slot;             /* memory slot index */
    __u32 flags;            /* KVM_MEM_LOG_DIRTY_PAGES | KVM_MEM_READONLY */
    __u64 guest_phys_addr;  /* base GPA of this region */
    __u64 memory_size;      /* size in bytes; 0 deletes the slot */
    __u64 userspace_addr;   /* HVA: address of the mmap'd backing buffer */
};
```

The `slot` field is an index into the kernel's memory slot table. Each slot describes one contiguous GPA-to-HVA mapping; the kernel supports many slots simultaneously, which is how production VMMs carve out distinct regions for DRAM, ROM, firmware, and MMIO holes. `flags` is almost always zero for a regular RAM slot; `KVM_MEM_LOG_DIRTY_PAGES` enables tracking for live migration, and `KVM_MEM_READONLY` makes the guest unable to write the range.

The LWN reference VMM reduces this to its minimum: one `mmap` call, one slot:

```
void *mem = mmap(NULL, 0x1000,
    PROT_READ | PROT_WRITE,
    MAP_SHARED | MAP_ANONYMOUS, -1, 0);

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

GPA `0x1000` is chosen so the guest starts executing at `CS:IP = 0x0000:0x1000` (flat physical `0x1000`) in 16-bit real mode. One page, one slot, one shot.

Firecracker does the same thing structurally but at a different scale. `build_microvm_for_boot()` in `src/vmm/src/builder.rs` calls `vm_resources.allocate_guest_memory()` before anything else, then `vm.register_dram_memory_regions(guest_memory)`, which wraps `KVM_SET_USER_MEMORY_REGION` in a loop over named slots. The guest-physical layout is defined in `src/vmm/src/arch/x86_64/layout.rs`: the kernel loads at `HIMEM_START = 0x100000` (1 MB), `boot_params` lives at the zero page `0x7000`, the command line sits at `0x20000`, and the IOAPIC MMIO window opens at `0xFEC00000`. The numbers are fixed constants in that file; every other component in Firecracker is built around them.

Job 2: Create vCPUs And Map kvm_run

`KVM_CREATE_VCPU` takes a single integer argument — the vCPU ID, which on x86 becomes the APIC ID. It returns a vCPU fd:

```
int vcpufd = ioctl(vmfd, KVM_CREATE_VCPU, 0);
```

The fd is nearly useless until the VMM maps the **`kvm_run` communication page**. This is the mechanism by which guest exit state crosses the kernel–userspace boundary without a copy: the kernel writes `exit_reason` and the exit-specific data directly into a page that the VMM can also read, via a shared mapping:

```
int mmap_size = ioctl(kvmfd, KVM_GET_VCPU_MMAP_SIZE, 0);
struct kvm_run *run = mmap(NULL, mmap_size,
    PROT_READ | PROT_WRITE, MAP_SHARED, vcpufd, 0);
```

`KVM_GET_VCPU_MMAP_SIZE` (`_IO(0xAE, 0x04)`) is called on the *system* fd — `/dev/kvm` — not on the vCPU fd. That surprises most first-time readers. The returned size is usually larger than `sizeof(struct kvm_run)` because the kernel stores transient per-vCPU scratch state in the same mapping immediately after the struct. `MAP_SHARED` is not optional; without it, the guest's writes to `exit_reason` never reach the VMM.

`struct kvm_run` contains an input half and an output half, separated by the `exit_reason` field:

```
struct kvm_run {
    /* inputs: VMM writes these before KVM_RUN */
    __u8  request_interrupt_window;
    __u8  immediate_exit;
    __u8  padding1[6];

    /* output: kernel writes this after each VM-exit */
    __u32 exit_reason;

    __u8  ready_for_interrupt_injection;
    __u8  if_flag;
    __u16 flags;
    __u64 cr8;
    __u64 apic_base;

    union {
        struct { /* KVM_EXIT_IO */
            __u8  direction;   /* 0 = IN, 1 = OUT */
            __u8  size;        /* 1, 2, or 4 bytes */
            __u16 port;
            __u32 count;
            __u64 data_offset; /* offset from kvm_run* to the data buffer */
        } io;
        struct { /* KVM_EXIT_MMIO */
            __u64 phys_addr;
            __u8  data[8];
            __u32 len;
            __u8  is_write;
        } mmio;
        struct { /* KVM_EXIT_FAIL_ENTRY */
            __u64 hardware_entry_failure_reason;
            __u32 cpu;
        } fail_entry;
        struct { /* KVM_EXIT_HYPERCALL */
            __u64 nr;
            __u64 args[6];
            __u64 ret;
        } hypercall;
        char padding[256];
    };
};
```

Setting `immediate_exit` to `1` before calling `KVM_RUN` — or from a separate thread while the vCPU is already running — forces the guest out as quickly as possible. The kernel checks it on every safe-to-exit point and returns `EINTR` to the calling thread. The `request_interrupt_window` field asks the kernel to exit as soon as the guest's interrupt flag is enabled, which is how VMMs inject interrupts without racily hitting a window where `IF = 0`.

Job 3: Load The Guest

With memory registered and a vCPU created, the VMM has to put executable code into guest memory and configure the CPU state to match. This job splits naturally into two parts: writing the guest image into the GPA range and setting the register file via `KVM_SET_REGS` and `KVM_SET_SREGS`.

The register setup is architecture-specific and fiddly. The LWN minimal VMM boots a tiny flat binary in real mode and sets registers directly:

```
struct kvm_sregs sregs;
ioctl(vcpufd, KVM_GET_SREGS, &sregs);
sregs.cs.base     = 0;
sregs.cs.selector = 0;
ioctl(vcpufd, KVM_SET_SREGS, &sregs);

struct kvm_regs regs = {
    .rip    = 0x1000,
    .rax    = 2,
    .rbx    = 2,
    .rflags = 0x2,     /* bit 1 is architecturally reserved-set */
};
ioctl(vcpufd, KVM_SET_REGS, &regs);
```

`rflags = 0x2` is not a quirk — x86 defines bit 1 of EFLAGS as permanently reserved-set. Failing to set it causes `KVM_EXIT_FAIL_ENTRY` before the first instruction retires.

A production VMM targeting a Linux guest does not poke registers in real mode. It implements the **Linux/x86 boot protocol**, current version **2.15** (introduced in kernel 5.5). The protocol specifies a `boot_params` structure (the "zero page") placed at a well-known GPA, populated with a `setup_header` that begins at offset `0x01F1` in the kernel image. The VMM must write `0xAA55` at `boot_params[0x1FE]` (the `boot_flag`), `0x53726448` ("HdrS") at `0x202`, `0xFF` in `type_of_loader`, and a physical pointer to the kernel command line in `cmd_line_ptr`. For `bzImage` format (boot protocol >= 2.00), the protected-mode kernel body loads at physical address `0x100000` when `LOADED_HIGH` (bit 0 of `loadflags`) is set; the 32-bit entry point expects `CS = 0x10` (4 GB flat), `DS = ES = SS = 0x18`, `%esi` pointing to `boot_params`, paging off, and interrupts off.

kvmtool chooses 16-bit real mode. Its `x86/kvm-cpu.c` sets:

```
kvm_regs.rip    = arch.boot_ip;   /* must be <= 65535 */
kvm_regs.rsp    = arch.boot_sp;
kvm_regs.rbp    = arch.boot_sp;
kvm_regs.rflags = 0x0000000000000002ULL;
```

All segment registers (CS, SS, DS, ES, FS, GS) receive the same `boot_selector`, and the base for each is computed with the standard real-mode left-shift:

```
static inline uint32_t selector_to_base(uint16_t sel) {
    return (uint32_t)sel << 4;
}
```

kvmtool also initializes MSRs via `KVM_SET_MSRS` (`_IOW(0xAE, 0x89, struct kvm_msrs)`): `MSR_IA32_SYSENTER_CS/ESP/EIP` are all zeroed, `MSR_IA32_TSC` is zeroed, and `MSR_IA32_MISC_ENABLE` has the `FAST_STRING` bit enabled. FPU state via `KVM_SET_FPU` (`_IOW(0xAE, 0x8d, struct kvm_fpu)`) is set to `fcw = 0x37f`, `mxcsr = 0x1f80` — the x87 control word and SSE control register their reset values.

Firecracker goes further still. `src/vmm/src/arch/x86_64/regs.rs` programs the vCPU into 64-bit long mode directly: CR0 has `PE` and `ET` set, CR4 has `PAE` set, the EFER MSR has both `LME` and `LMA` set, the CS descriptor is built from `gdt_table[1]` (GDT index 1) giving selector `0x08` — a 64-bit execute/read code segment with `L = 1` — and the GDT is loaded at GPA `0x500`. Firecracker also supports **PVH boot** (32-bit protected mode with the PVH magic `0x336EC578`), which skips real mode entirely and hands control to the kernel at a different entry point. The `KVM_SET_CPUID2` (`_IOW(0xAE, 0x90, struct kvm_cpuid2)`) call configures which CPU features the guest sees, and `KVM_SET_SIGNAL_MASK` (`_IOW(0xAE, 0x8b, struct kvm_signal_mask)`) optionally controls which signals interrupt `KVM_RUN` on the vCPU thread.

Job 4: Run The Loop

`KVM_RUN` (`_IO(0xAE, 0x80)`) is a no-argument vCPU `ioctl`. It blocks the calling thread, switches the physical CPU into VMX or SVM guest mode, and does not return until a VM-exit occurs. On success it returns `0`; `kvm_run->exit_reason` identifies why the guest exited. On `EINTR` (a signal arrived before or during guest execution) it returns `-1` with `errno = EINTR`.

Every production VMM is a direct elaboration of this loop:

```
while (1) {
    ioctl(vcpufd, KVM_RUN, NULL);
    switch (run->exit_reason) {
    case KVM_EXIT_HLT:
        return 0;                   /* clean shutdown */
    case KVM_EXIT_IO:
        if (run->io.direction == KVM_EXIT_IO_OUT
            && run->io.size == 1
            && run->io.port == 0x3f8
            && run->io.count == 1)
            putchar(*(((char *)run) + run->io.data_offset));
        break;
    case KVM_EXIT_FAIL_ENTRY:
        errx(1, "KVM_EXIT_FAIL_ENTRY: 0x%llx\n",
             run->fail_entry.hardware_entry_failure_reason);
    case KVM_EXIT_INTERNAL_ERROR:
        errx(1, "KVM_EXIT_INTERNAL_ERROR: suberror = 0x%x\n",
             run->internal.suberror);
    }
}
```

`KVM_EXIT_HLT` (value `5`) means the guest executed an `HLT` instruction and has no pending interrupt to wake it. For the minimal VMM this signals a clean exit; for a real VMM it means putting the vCPU thread to sleep until an interrupt arrives. `KVM_EXIT_FAIL_ENTRY` (value `9`) is distinct and serious: KVM tried to enter guest mode but the hardware rejected the VMCS or VMCB state. `fail_entry.hardware_entry_failure_reason` encodes the hardware VM-exit reason that the processor reported when rejecting VM-entry, and it usually means the VMM set an illegal register combination during job 3. `KVM_EXIT_INTERNAL_ERROR` (value `17`) means KVM itself detected an inconsistency; it is always fatal.

`data_offset` is a **byte offset from the start of the `kvm_run` struct**, not a pointer. The data buffer sits inside the same mmap'd region as the struct, past the fixed header. Casting to `(char *)run + run->io.data_offset` is correct; dereferencing `run->io.data_offset` as a pointer is a common first-time mistake that produces a segfault at address `~80`.

The full `KVM_EXIT_*` vocabulary used in production is broader. A selection of the values in `include/uapi/linux/kvm.h` worth knowing:

Constant	Value	Meaning
`KVM_EXIT_UNKNOWN`	0	Hardware exit reason unrecognized
`KVM_EXIT_IO`	2	Guest PIO (`IN` / `OUT` instruction)
`KVM_EXIT_HYPERCALL`	3	Guest hypercall (`VMCALL` / `VMMCALL`)
`KVM_EXIT_HLT`	5	Guest executed `HLT` with no pending interrupt
`KVM_EXIT_MMIO`	6	Guest accessed an unmapped GPA
`KVM_EXIT_SHUTDOWN`	8	Triple fault or guest-requested shutdown
`KVM_EXIT_FAIL_ENTRY`	9	Hardware refused VM-entry; see `fail_entry.hardware_entry_failure_reason`
`KVM_EXIT_INTR`	10	Signal arrived during `KVM_RUN`
`KVM_EXIT_INTERNAL_ERROR`	17	KVM internal consistency error; always fatal
`KVM_EXIT_X86_RDMSR`	29	Guest `RDMSR` with no in-kernel handler
`KVM_EXIT_X86_WRMSR`	30	Guest `WRMSR` with no in-kernel handler
`KVM_EXIT_MEMORY_FAULT`	39	Guest accessed GPA with no valid mapping

kvmtool's `kvm-cpu.c` handles `KVM_EXIT_UNKNOWN`, `KVM_EXIT_DEBUG`, `KVM_EXIT_IO`, `KVM_EXIT_MMIO`, `KVM_EXIT_INTR`, `KVM_EXIT_SHUTDOWN`, and `KVM_EXIT_SYSTEM_EVENT`. It also drains coalesced MMIO (via `KVM_COALESCED_MMIO_PAGE_OFFSET`) both before and after the standard MMIO exit path — a performance optimization that batches MMIO writes from the guest to reduce the number of true VM-exits.

Job 5: Emulate Devices

Most VM-exits are the guest asking the VMM to do something on its behalf. The two most common forms are PIO (`KVM_EXIT_IO`) and MMIO (`KVM_EXIT_MMIO`).

PIO exits arrive when the guest executes an `IN` or `OUT` instruction. The `io` sub-struct in `kvm_run` carries everything needed:

- `direction` : `0` for `IN` (guest reads from port), `1` for `OUT` (guest writes to port)
- `size` : data width in bytes — `1` , `2` , or `4`
- `port` : the x86 I/O port number
- `count` : number of consecutive operations (for `INS` / `OUTS` string instructions)
- `data_offset` : byte offset within the `kvm_run` struct to the data buffer

For `KVM_EXIT_IO_OUT`, the data is already in the buffer when the VMM reads it; the VMM routes the write to the appropriate emulated device. For `KVM_EXIT_IO_IN`, the VMM writes the device's response into the buffer and then re-enters `KVM_RUN` — the kernel delivers those bytes to the `IN` instruction as if they came from real hardware.

MMIO exits arrive when the guest accesses a GPA that has no registered memory slot. The `mmio` sub-struct carries:

- `phys_addr`: the GPA of the access
- `data[8]`: up to 8 bytes of data
- `len`: the access size in bytes
- `is_write`: `1` if the guest is writing, `0` if reading

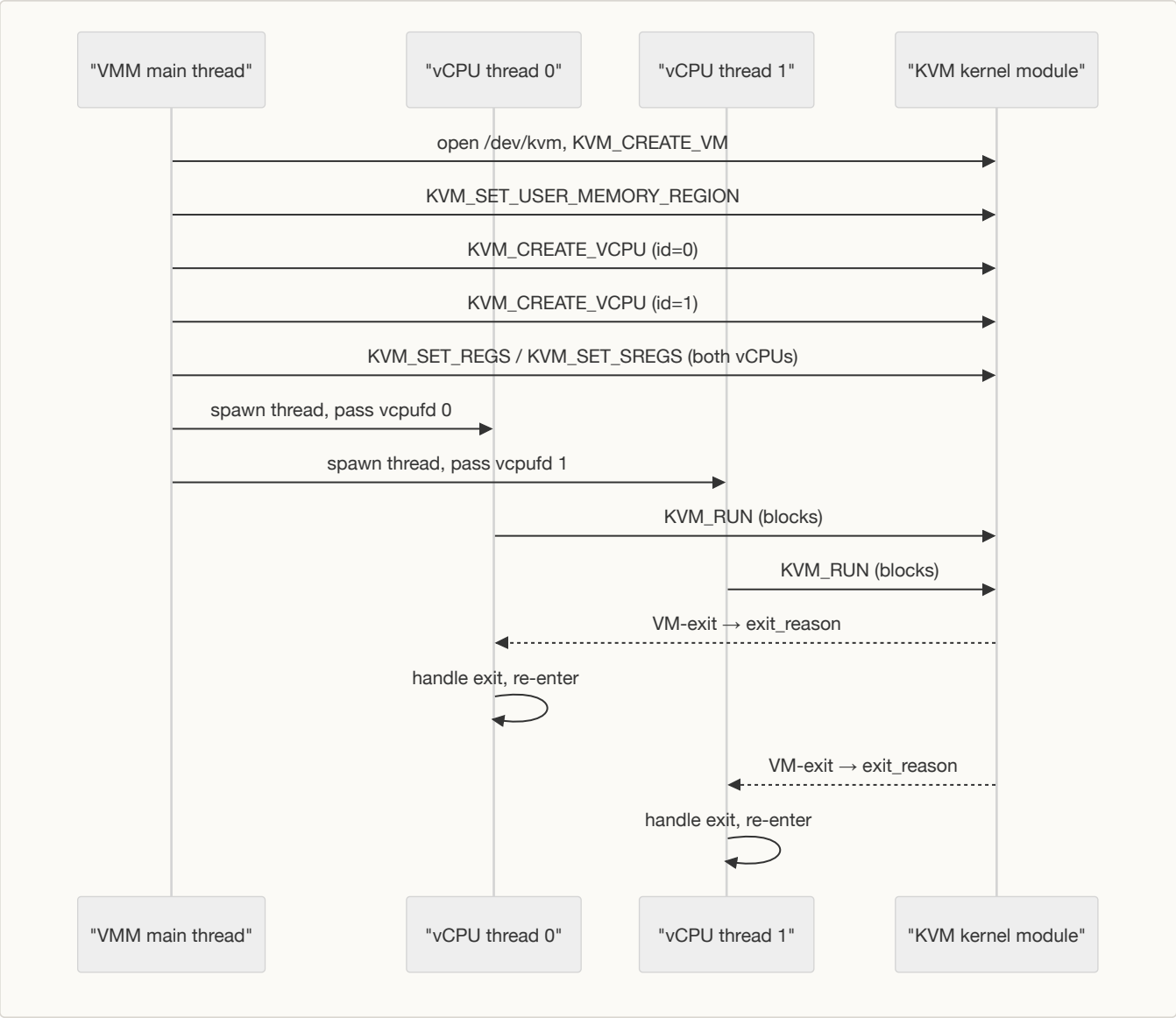
The VMM dispatches `phys_addr` to the emulated device that owns that GPA range. Firecracker uses virtio-MMIO for every device — there is no PCI bus — so its MMIO handler routes accesses in the 32-bit MMIO region starting at `0xC0000000` (`MMIO32_MEM_START` in `layout.rs`) to the appropriate virtio backend, reads or writes the virtio queue registers, and re-enters.

The emulation path is also where interrupt injection happens. After servicing an MMIO or PIO write that updates a device's status (say, a virtio queue kick), the VMM may need to deliver an interrupt back to the guest CPU. With an in-kernel irqchip (`KVM_CREATE_IRQCHIP`), Firecracker routes interrupts through the IOAPIC GSI table (`KVM_SET_GSI_ROUTING`) rather than directly manipulating the APIC — the in-kernel implementation handles the interrupt delivery atomically with the next `KVM_RUN` re-entry. The minimal VMM has none of this; the virtio story and interrupt injection are the subjects of later chapters.

The vCPU Thread Model

`KVM_RUN` blocks its calling thread for the entire duration of guest execution. That constraint is the whole vCPU thread model: one host thread per guest CPU, each calling `KVM_RUN` in a loop on its own vCPU fd. The kernel documentation is explicit: "To run a multi-CPU VM, the user-space process must spawn multiple threads, and call `KVM_RUN` for different virtual CPUs in different threads." The kernel schedules those POSIX threads across physical cores, so a 2-vCPU guest can genuinely execute two streams of guest code on two physical CPUs simultaneously.

The sequence from process start to a running multi-vCPU guest follows this shape:



The kernel documentation notes that vCPU `ioctl`s should be issued from the same thread that created the vCPU. Migrating a vCPU fd to a different thread is not forbidden, but the first `ioctl` from the new thread incurs TLB and scheduler effects as the kernel re-pins the vCPU to the new physical CPU.

Interrupting A Running vCPU

When the VMM needs to pull a vCPU out of guest mode — to inject an interrupt, to handle an API request, to stop the VM — it cannot simply call a function, because the vCPU thread is blocked inside the kernel. The mechanism has three parts:

1. Write `1` to `kvm_run->immediate_exit` (a `u8` field in the mmap'd struct). The kernel checks this flag at every safe VM-exit point and returns `EINTR` to the thread when it is set.
2. Issue a memory fence to ensure the write is visible before the signal arrives.

3. Send a signal to the vCPU thread. The signal causes `KVM_RUN` to return `-1` with `errno = EINTR` if the vCPU is in guest mode, or prevents the next `KVM_RUN` call from entering guest mode if the signal arrives first.

The vCPU thread on the other side clears `immediate_exit` back to `0` and checks for pending events before re-entering the loop.

An alternative is `KVM_SET_SIGNAL_MASK` (`_IOW(0xAE, 0x8b, struct kvm_signal_mask)`), which lets the VMM declare exactly which signals interrupt `KVM_RUN`. Any unmasked signal that arrives during the `ioctl` causes it to return `-EINTR`. This is useful when the process has other signal handlers that should not disturb the run loop.

The kernel also has an internal wake path, `kvm_vcpu_kick()`, used when one component of the KVM code needs to stop a vCPU that is executing inside the guest. `kvm_vcpu_kick()` sends an inter-processor interrupt (IPI) to the physical CPU running the guest, causing a VM-exit, after which the kernel checks the `vcpu->requests` bitmap (set by `kvm_make_request()`) before allowing re-entry.

Firecracker's Thread Categories

Firecracker structures its threads into three roles:

- **API thread**: runs an HTTP server on a Unix domain socket, handling `PUT` / `GET` / `PATCH` requests that configure the microVM.
- **VMM thread**: owns device emulation, MMIO dispatch, rate limiting, and IRQ injection.
- **N vCPU threads**: one per guest core, named `fc_vcpu 0`, `fc_vcpu 1`, and so on. Firecracker supports up to 32 vCPUs.

Each vCPU thread is spawned in `Vcpu::start_threaded()` inside `src/vmm/src/vstate/vcpu.rs`:

```
thread::Builder::new()
    .name(format!("fc_vcpu {}", self.kvm_vcpu.index))
    .spawn(move || {
        self.register_kick_signal_handler();
        barrier.wait();
        self.run(seccomp_filter)
    })
```

The thread registers a handler for `SIGRTMIN + 0` (Firecracker's `VCPU_RTSIG_OFFSET = 0`) as its kick signal, waits on a barrier so all threads synchronize at start, and then enters `self.run()`. That function implements a state machine: the initial state is `paused`; the `running` state calls `run_emulation()` in a tight loop, which calls `self.kvm_vcpu.fd.run()` — the `kvm-ioctls` crate's safe wrapper around `ioctl(vcpufd, KVM_RUN, NULL)`. When `KVM_RUN` returns `EINTR`, Firecracker sets `immediate_exit = 0` and returns `VcpuEmulation::Interrupted`, triggering a check of the mpsc channel the VMM thread

uses to send events. This is the seam between the run loop and everything else: the vCPU thread is single-purpose and reactive; complexity lives in the VMM thread and crosses to the vCPU thread only through the event channel and `kvm_run`.

Firecracker wraps all KVM kernel types through the `rust-vmm` crate ecosystem: `kvm-bindings` provides Rust FFI structs generated from kernel headers, and `kvm-ioctls` provides safe wrappers — `Kvm`, `VmFd`, `VcpuFd` — that mirror the three-level fd hierarchy exactly. `VcpuFd::run()` calls the `ioctl`, reads `exit_reason` from the mmap'd struct, and returns a `VcpuExit<'_>` enum variant; on failure it checks specifically for `KVM_EXIT_MEMORY_FAULT` (value `39`) before returning an error, because that exit requires distinct handling from other kernel errors.

Five Jobs, One Picture

The five jobs are not parallel: each depends on the last. You cannot create a vCPU before you have a VM fd, cannot run the loop before you have loaded the guest, cannot emulate devices before the loop is running to deliver exits to you.

Jobs 1 through 3 are one-time setup. Jobs 4 and 5 repeat in a tight loop for every cycle of guest time. The boundary between them — `KVM_RUN` returning, `exit_reason` being read, a handler firing, `KVM_RUN` being called again — is the innermost loop of every VMM, and its latency is the floor on device emulation performance.

The LWN example covers this loop in about 200 lines of C. kvmtool adds real-mode boot, MSR initialization, and a proper exit handler in a few thousand. Firecracker adds a security sandbox, virtio devices, and a three-thread architecture on top of the same five jobs, in roughly 30,000 lines of Rust. The mechanism does not change. The policy around it does.

Sources And Further Reading

- KVM API documentation: https://docs.kernel.org/virt/kvm/api.html
- Linux `kvm.h` ioctl definitions: https://sites.uclouvain.be/SystInfo/usr/include/linux/kvm.h.html
- torvalds/linux `include/uapi/linux/kvm.h`: https://github.com/torvalds/linux/blob/master/include/uapi/linux/kvm.h
- LWN "Using the KVM API": https://lwn.net/Articles/658511/
- Linux/x86 boot protocol: https://www.kernel.org/doc/html/latest/arch/x86/boot.html

- KVM vCPU requests documentation: https://www.kernel.org/doc/html/v5.7/virt/kvm/vcpu-requests.html
- kvmtool `x86/kvm-cpu.c` (register setup): https://github.com/clearlinux/kvmtool/blob/master/x86/kvm-cpu.c
- kvmtool `kvm-cpu.c` (exit handler): https://github.com/kvmtool/kvmtool/blob/master/kvm-cpu.c
- Firecracker design doc: https://github.com/firecracker-microvm/firecracker/blob/main/docs/design.md
- Firecracker `src/vmm/src/builder.rs`: https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/builder.rs
- Firecracker `src/vmm/src/arch/x86_64/layout.rs`: https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/arch/x86_64/layout.rs
- Firecracker `src/vmm/src/arch/x86_64/regs.rs`: https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/arch/x86_64/regs.rs
- Firecracker `src/vmm/src/vstate/vcpu.rs`: https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/vstate/vcpu.rs
- rust-vmm/kvm-ioctls `vcpu.rs`: https://github.com/rust-vmm/kvm-ioctls/blob/main/kvm-ioctls/src/ioctls/vcpu.rs
- kvm-ioctls API docs: https://docs.rs/kvm-ioctls/latest/kvm_ioctls/

Chapter 10: Booting A Guest Kernel

The previous chapter left the vCPU in reset state — halted, expecting firmware to run first. In a conventional VM that firmware is SeaBIOS or OVMF. It runs POST, scans PCI, builds an E820 memory map, loads GRUB, which reads a config file, which decompresses a kernel, which eventually calls `start_kernel`. The whole sequence takes hundreds of milliseconds and adds two software layers that the guest can never directly observe or control. For a microVM that is designed to start in under 125 ms and whose kernel image is known at build time, firmware is pure overhead. This chapter is about eliminating it entirely.

The question the chapter answers is precise: given a KVM file descriptor, a guest physical address space, and a kernel image on disk, what is the exact machine state the VMM must establish before issuing `KVM_RUN` so that the kernel's first instruction executes correctly with no BIOS, bootloader, or decompressor?

The answer splits into two branches depending on the kernel image format and build configuration. Both branches end at the same place — the kernel's `start_kernel` in `init/main.c` — but they take different paths through the x86_64 architecture's mode hierarchy to get there. Understanding them requires knowing what the kernel expects the CPU and memory to look like at its entry point, which means reading the boot protocol directly.

The Two Boot Paths

Before the Linux boot protocol existed, every bootloader assumed a different entry state and every kernel shipped a compatibility shim that tried to detect which one it was running under. The protocol, formalized starting with version `0x0200` in Linux 1.3.73, replaced that chaos with a contract: the VMM or bootloader fills in a structured record in guest memory, places its guest-physical address in a well-known register, and jumps to a well-known offset in the kernel image. The kernel reads the record at startup and learns everything it cannot detect directly — where its initrd lives, what the command line says, what physical memory ranges are available.

There is a second contract, newer and narrower in scope: the PVH boot ABI, which originated in the Xen project and entered mainline Linux in version 5.0. PVH makes a different tradeoff. It sacrifices the 64-bit paging setup that the Linux protocol requires the loader to perform, in exchange for a simpler CPU entry state — 32-bit protected mode with paging disabled — and substitutes a smaller data structure, `hvm_start_info`, for the Linux `boot_params`. The kernel's PVH entry point then transitions itself to 64-bit long mode. Any x86_64 microVM VMM has to know both paths, because which one applies depends on which ELF notes the kernel image contains.

Firecracker on x86_64 implements both. It inspects the loaded ELF kernel for a `XEN_ELFNOTE_PHYS32_ENTRY` note (type 18, name `"Xen"`); if the note is present, Firecracker uses PVH as the preferred path. If the note is absent, it falls back to the 64-bit Linux protocol. The choice is made

per image at every boot. The rest of this chapter covers the Linux protocol in depth first, then the PVH path as a variation, because most existing documentation — and most existing kernels — leads with the Linux protocol.

vmlinux vs. bzImage

The image format determines how the VMM extracts both the entry point and the setup data.

`vmlinux` is the raw ELF produced by the kernel build. It is statically linked, 64-bit, uncompressed, and carries the full ELF header that any standard ELF loader can parse. The VMM loads its `PT_LOAD` segments directly into guest RAM and reads the ELF `e_entry` field to find `startup_64`, the first kernel instruction, in `arch/x86/kernel/head_64.S`. No decompression step runs, because there is nothing to decompress.

`bzImage` is a different animal. It is a self-extracting archive: the first `(setup_sects + 1) × 512` bytes are a real-mode setup blob in 16-bit code, and the bytes after that are the protected-mode kernel compressed according to whatever `CONFIG_KERNEL_*` option was selected — gzip, LZMA, or zstd. A bootloader that ingests a bzImage has to extract the `setup_header` from file offset `0x01F1`, potentially run the real-mode stub to set up the environment, and let the embedded decompressor unpack the kernel before any kernel code runs. Measured against a direct ELF load, Stefano Garzarella (Red Hat) observed approximately 78 ms for the compressed bzImage path in QEMU 4.0, versus approximately 10 ms for PVH entry via vmlinux in QEMU 4.0 — a 7.8x difference, primarily attributable to the in-guest decompression and real-mode phases.

Firecracker requires an uncompressed ELF `vmlinux` on x86_64. It does not support bzImage. Build the guest kernel with `make vmlinux`. On aarch64, Firecracker instead requires the PE-format `Image` file produced by `make Image`.

The rust-vmm `linux-loader` crate, which Firecracker uses internally, supports three formats: raw ELF on x86_64 (for both the Linux protocol and PVH), bzImage on x86_64, and PE `Image` on aarch64/riscv64. For an ELF kernel, `linux-loader` returns only the kernel entry address. For bzImage it additionally extracts the `setup_header` from file offset `0x01F1` and returns it alongside the entry address. In both cases the VMM is responsible for constructing `struct boot_params` from that header and from its own knowledge of the memory layout.

The Setup Header and the Zero Page

The boot protocol's central data structure is `struct boot_params`, a 4096-byte, packed C struct that the Linux kernel defines in `arch/x86/include/uapi/asm/bootparam.h`. The kernel documentation calls the page it occupies the **zero page**, because early boot code expects to find it at the start of the setup segment — historically at physical address 0. In a direct-boot microVM, the VMM places it at whatever guest-physical address it chooses, as long as it communicates that address to the kernel via `RSI` at entry.

Embedded within `boot_params` at offset `0x01F1` is `struct setup_header`, roughly 144 bytes of protocol fields. For a bzImage, the setup header is copied verbatim from the image file at the same byte offset. For an ELF `vmlinux`, the VMM fills in a minimal synthetic header, because the fields the kernel cares about at direct-boot time are only those the VMM itself controls — loader type, command-line address, initrd address, and memory alignment.

The VMM identifies a valid kernel image by checking two magic numbers that `arch/x86/boot/header.S` writes at link time:

```
boot_flag: .word 0xAA55
header:    .ascii "HdrS"        # 0x5372_6448 little-endian
```

`boot_flag` at image offset `0x01FE` must equal `0xAA55` — a repurposed MBR signature. `header` at offset `0x0202` must equal `0x5372_6448` (ASCII `HdrS`). The rust-vmm bzImage loader rejects any image where `boot_header.header != 0x5372_6448`. The protocol version is a two-byte little-endian value at offset `0x0206`: `(major << 8) | minor`. Current kernels report `0x020F` (protocol 2.15), introduced in Linux 5.5.

Protocol versions matter because they gate individual fields. A VMM that writes `cmd_line_ptr` assumes at least protocol 2.02 (Linux 2.4.0-test3-pre3). The `cmdline_size` field at `0x0238` requires protocol 2.06 (Linux 2.6.22). The `setup_data` linked-list pointer at `0x0250` — used to pass ACPI tables, DTB blobs, and EFI memory maps without extending `boot_params` itself — requires protocol 2.09 (Linux 2.6.26). The `xloadflags` field at `0x0236`, whose bit 0 (`XLF_KERNEL_64`) indicates a valid 64-bit entry point at `load_addr + 0x200`, requires protocol 2.12 (Linux 3.8). Any modern kernel the microVM stack would care about supports at least 2.12; protocol 2.15 kernels are the current baseline.

The fields a VMM must fill in `setup_header` for a direct 64-bit boot are a small subset of the full spec:

Offset	Field	Required value	Meaning
`0x0210`	`type_of_loader`	`0xFF`	No registered bootloader ID
`0x01FE`	`boot_flag`	`0xAA55`	Sanity sentinel
`0x0202`	`header`	`0x5372_6448`	Protocol magic
`0x0228`	`cmd_line_ptr`	32-bit GPA	Guest-physical address of command line
`0x0238`	`cmdline_size`	byte count	Length of command line excluding null
`0x0230`	`kernel_alignment`	`0x0100_0000`	16 MiB alignment (Firecracker value)
`0x0218`	`ramdisk_image`	32-bit GPA	Initrd start; zero if none
`0x021C`	`ramdisk_size`	byte count	Initrd size in bytes; zero if none

Firecracker sets `type_of_loader` to `0xFF`, the catch-all value `KERNEL_LOADER_OTHER` defined by the boot protocol for VMMs and loaders without registered IDs.

Populating the Zero Page

The full `boot_params` struct is 4096 bytes. Most of it is zeroed. Beyond the `setup_header` section, two fields carry information the kernel cannot derive independently: the ACPI RSDP address and the physical memory map.

Firecracker places `boot_params` at guest-physical address `0x7000` (`ZERO_PAGE_START`). It fills `acpi_rsdp_addr` at `boot_params[0x070]` with the address `0x000E_0000`, where the ACPI tables were built. The memory map lives at `boot_params[0x2D0]` as an array of up to 128 `boot_e820_entry` structs — each 20 bytes: a `u64` start address, a `u64` size, and a `u32` type. The count of valid entries goes at `boot_params[0x1E8]` as a `u8`. Type 1 is `E820_RAM` (usable); type 2 is `E820_RESERVED`.

Firecracker's `configure_64bit_boot()` in `src/vmm/src/arch/x86_64/mod.rs` builds the table with four classes of entry:

1. `[0x0000_0000, 0x9FC00)` → type 1: usable RAM below the Extended BIOS Data Area.
2. `[0x9FC00, 0x9FC00 + 0x40400)` → type 2: reserved region covering the EBDA, the MP table, and the ACPI area.
3. `[PCI_MMCONFIG_START, PCI_MMCONFIG_START + 256 MiB)` → type 2: the PCIe ECAM window.
4. `[max(0x10_0000, region.start), region.end)` → type 1 per DRAM region: usable RAM from 1 MiB upward.

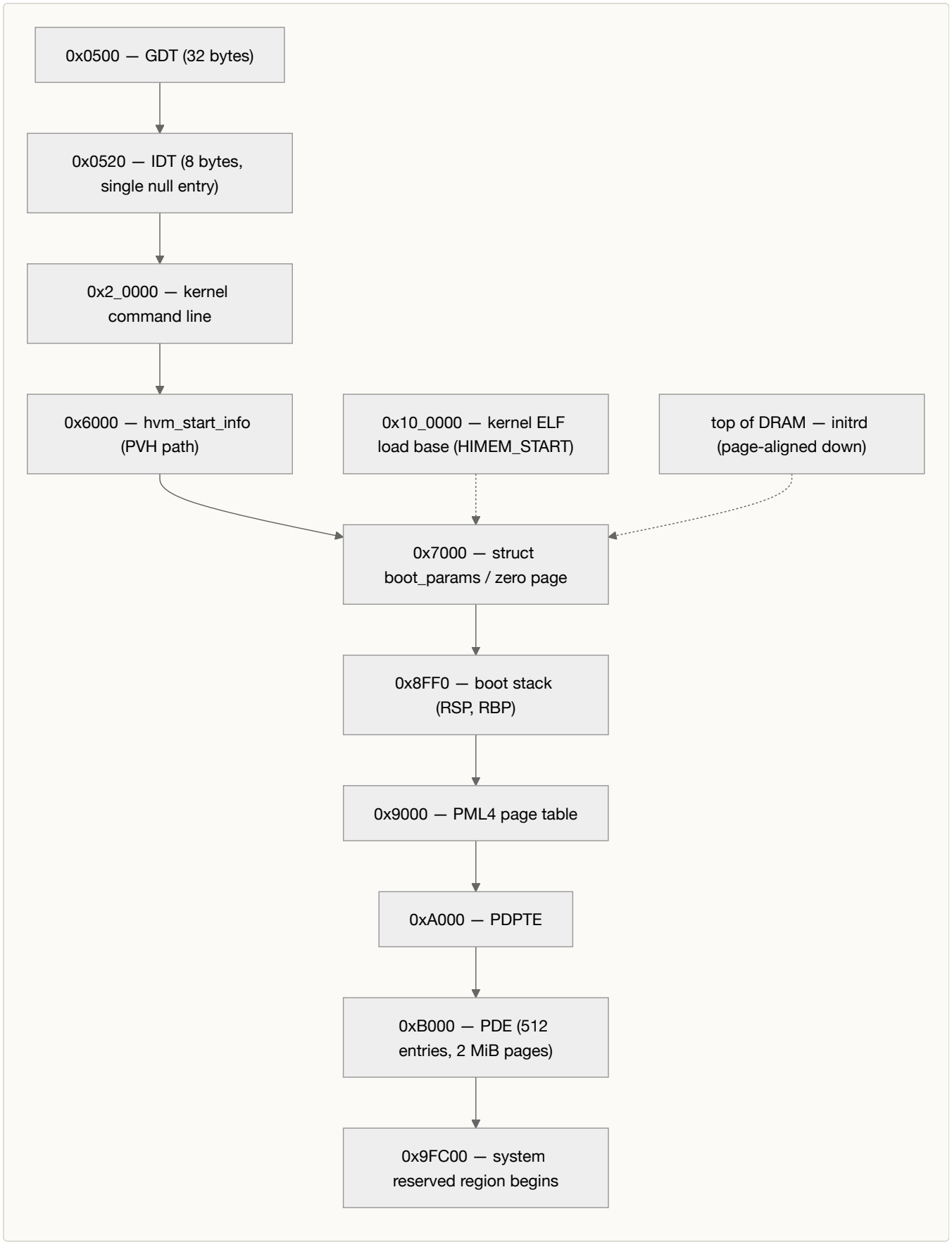
That table is what the kernel reads at `startup_64` to construct its own memory model. It replaces the E820 query that a real BIOS would answer.

The complete `boot_params` assembly that Firecracker writes looks like this:

Once written, this page is the only document the kernel has to understand its environment. Everything that BIOS POST and GRUB would have discovered or constructed over hundreds of milliseconds is expressed in 4096 bytes assembled by the VMM in microseconds.

Placing the Kernel, Initrd, and Command Line

The address constants for x86_64 in Firecracker are defined in `src/vmm/src/arch/x86_64/layout.rs`. A tour of the low guest-physical address space shows how tightly packed the boot data is:


0x0500 — GDT (32 bytes)
0x0520 — IDT (8 bytes, single null entry)
0x2_0000 — kernel command line
0x6000 — hvm_start_info (PVH path)
0x10_0000 — kernel ELF load base (HIMEM_START)
top of DRAM — initrd (page-aligned down)
0x7000 — struct boot_params / zero page
0x8FF0 — boot stack (RSP, RBP)
0x9000 — PML4 page table
0xA000 — PDPTE
0xB000 — PDE (512 entries, 2 MiB pages)
0x9FC00 — system reserved region begins

The kernel loads at `HIMEM_START = 0x0010_0000` (1 MiB). Firecracker's `linux-loader` maps ELF `PT_LOAD` segments starting at that address, respecting the `kernel_alignment` of `0x0100_0000` (16 MiB) that the header declares. The kernel must be placed on a 16 MiB boundary at or above `HIMEM_START`.

The command line is a null-terminated C string at `CMDLINE_START = 0x0002_0000` (128 KiB), with a maximum length of 2048 bytes including the null terminator. `setup_header.cmd_line_ptr` receives `0x0002_0000` as a 32-bit guest-physical address. `setup_header.cmdline_size` receives the length of the specific command line string the VMM is providing, excluding the null terminator. This is distinct from the `cmdline_size` field in the kernel image's own setup header (protocol 2.06+), which declares the maximum command line length the kernel will accept; the VMM writes the actual string length, bounded by that capacity.

The initrd, if present, is placed at the top of the first DRAM region, page-aligned downward: `align_down(lowmem_end - initrd_size, PAGE_SIZE)`. This keeps the initrd as high as possible to avoid colliding with the kernel's own data. `setup_header.ramdisk_image` receives the resulting 32-bit guest-physical address; `setup_header.ramdisk_size` receives the byte count. The protocol defines `initrd_addr_max` (offset `0x022C`, added in protocol 2.03) as the highest address the initrd's last byte may occupy; the default in `arch/x86/boot/header.S` is `0x7FFF_FFFF` (just below 2 GiB), and a VMM that places the initrd above that limit must update the field or the kernel will reject it.

`KVM_TSS_ADDRESS = 0xFFFB_D000` is a special case. Before any vCPU can run, the VMM must issue `KVM_SET_TSS_ADDR` with this guest-physical address on Intel VMX hosts. KVM uses a hidden 3-page TSS region at this address to support its internal real-mode emulation, even though Firecracker never enters real mode. The companion call `KVM_SET_IDENTITY_MAP_ADDR` places a one-page identity-map page adjacent to the TSS. Both are VM-level ioctls. `KVM_SET_IDENTITY_MAP_ADDR` must be called before any `KVM_CREATE_VCPU`; the KVM API docs explicitly state it fails if a vCPU already exists. `KVM_SET_TSS_ADDR` has no documented ordering constraint relative to vCPU creation.

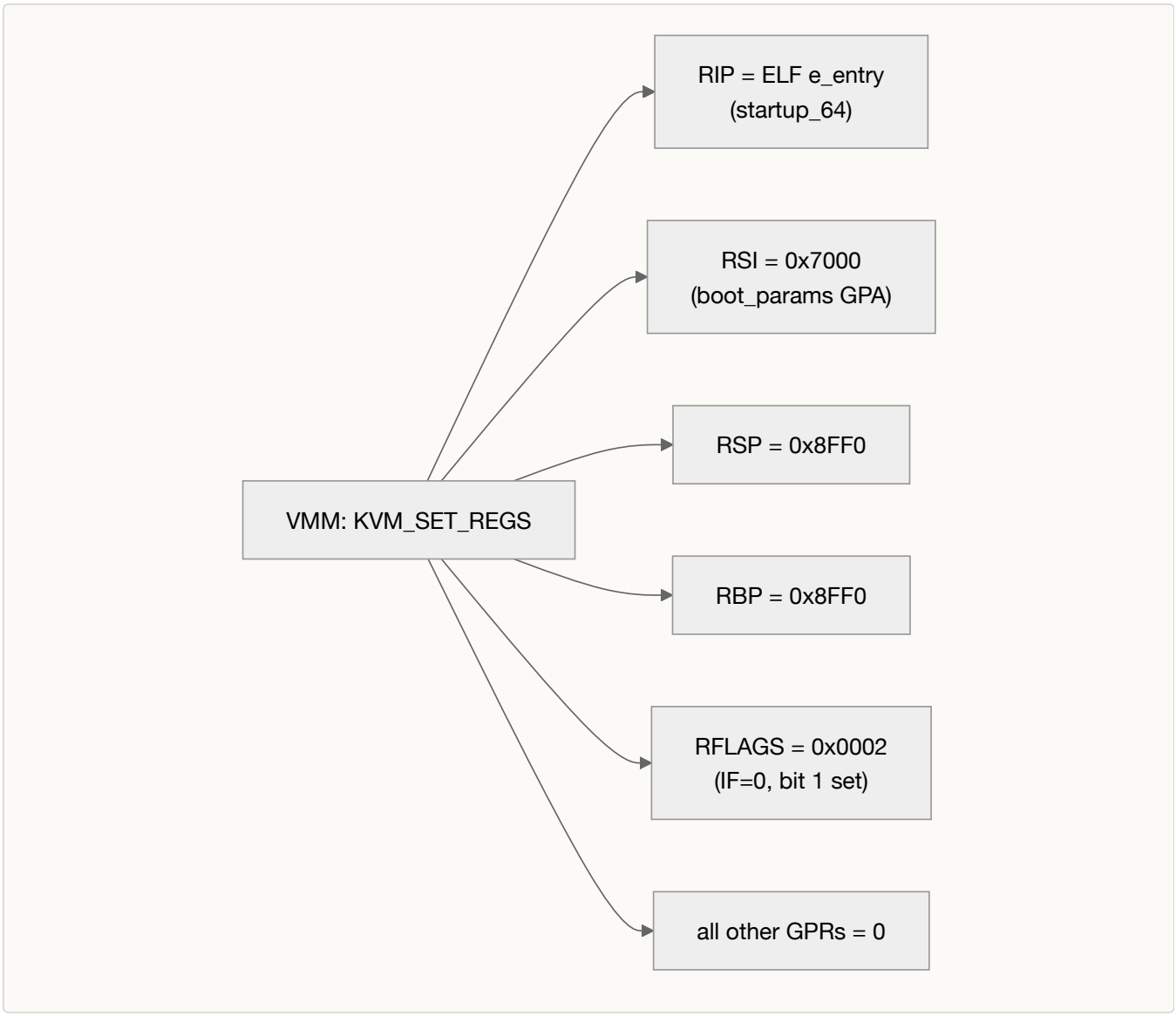
> ***Root required.*** *The `/dev/kvm` device is accessible only to members of the `kvm` group (or root). Both `KVM_SET_TSS_ADDR` and `KVM_SET_IDENTITY_MAP_ADDR` modify guest-physical address space. Running them on a production host requires the same permissions as any other KVM operation.*

CPU State at the 64-bit Entry Point

The Linux 64-bit boot protocol specifies an entry state that is already in long mode with paging active. This is the fundamental difference from a traditional bootloader, which enters the kernel in 32-bit protected mode and leaves mode switching to the kernel's own startup code. The 64-bit direct-boot protocol hands off in the mode the kernel will run in, making the kernel's `startup_64` path a simpler target.

The VMM programs this state with three ioctls on the vCPU file descriptor: `KVM_SET_REGS` for general-purpose registers, `KVM_SET_SREGS` for segment registers and control registers, and `KVM_SET_FPU` for floating-point state. All three must be issued before the first `KVM_RUN` on the vCPU.

General-Purpose Registers



`RIP` is the ELF `e_entry` field, which resolves to the `startup_64` symbol in `arch/x86/kernel/head_64.S`. This value is not a fixed numeric address — it depends on kernel configuration and, when KASLR is active, on randomization. The VMM reads it from the ELF header; it does not hardcode it.

`RSI` is the ABI contract: it must contain the 32-bit guest-physical address of `struct boot_params`. `startup_64` reads `RSI` as its first act. Any other value is undefined behavior from the kernel's point of view.

`RFLAGS` is `0x0000_0000_0000_0002`. Bit 1 is always 1 by architectural definition; `IF` (bit 9) is 0, leaving interrupts disabled until the kernel enables them during `start_kernel`. All other GPRs are zero.

Control Registers and Segment State

The VMM programs control registers and segment descriptors via `KVM_SET_SREGS`. The required values are:

Register	Value	Meaning
`CR0`	`PE | ET | PG`	Protected mode, extension type, paging enabled
`CR3`	`0x9000`	Guest-physical address of the boot PML4
`CR4`	existing `| PAE (0x20)`	Physical Address Extension for 4-level paging
`EFER`	existing `| LME (0x100) | LMA (0x400)`	Long Mode Enable and Long Mode Active

`PE` is bit 0 of `CR0` (`0x1`), `ET` is bit 4 (`0x10`), and `PG` is bit 31 (`0x8000_0000`). All three must be set simultaneously — turning on paging while not in protected mode is a fault. `PAE` is `CR4` bit 5 (`0x20`); 64-bit paging requires it. `LME` and `LMA` together in `EFER` signal that the CPU is in long mode and that the current `CS` descriptor is 64-bit. KVM validates these relationships and will refuse to enter the guest if they are inconsistent.

The GDT lives at guest-physical address `0x500`, with a 32-byte limit. Firecracker writes four entries in `src/vmm/src/arch/x86_64/gdt.rs`:

Index	Selector	Flags	Purpose
0	—	`0x0000`	NULL descriptor (mandatory first entry)
1	`0x08`	`0xA09B`	64-bit code: L=1, G=1, P=1, DPL=0, type=0xB
2	`0x10`	`0xC093`	Data: G=1, DB=1, P=1, DPL=0, type=0x3
3	`0x18`	`0x808B`	TSS: P=1, type=0xB (busy 32-bit TSS)

`sregs.cs` uses selector `0x08` (index 1, 64-bit code). `sregs.ds`, `es`, `fs`, `gs`, and `ss` all use selector `0x10` (index 2, data). `sregs.tr` uses selector `0x18` (index 3, TSS). The IDT at `0x520` is a single null 8-byte entry with `sregs.idt.limit = 7`; the kernel will install its own interrupt handlers early in `start_kernel`.

The Linux boot protocol documentation names these selectors `__BOOT_CS` (`0x10`) and `__BOOT_DS` (`0x18`), referring to positions 2 and 3 in the protocol's own GDT layout. Firecracker uses positions 1 and 2 (selectors `0x08` and `0x10`) instead — a harmless divergence, because KVM validates the descriptor attributes (64-bit code segment, flat data segment) rather than the selector values themselves.

Boot-Time Page Tables

The boot page tables at `PML4_START = 0x9000` create a minimal identity map covering guest-virtual `[0, 1 GiB)` using 2 MiB pages. Three levels suffice:

- **PML4** at `0x9000`: one entry pointing to the PDPTE — `0xA003` (address `0xA000` with present and writable bits set).
- **PDPTE** at `0xA000`: one entry pointing to the PDE array — `0xB003` (address `0xB000` with present and writable bits set).
- **PDE** at `0xB000`: 512 entries. Entry `i` is `(i << 21) | 0x83`: the 2 MiB page at physical address `i × 2 MiB`, with PS (bit 7), writable (bit 1), and present (bit 0) set.

This maps GPA 0 through 1 GiB – 2 MiB with a 1:1 virtual-to-physical correspondence. The boot protocol requires the kernel's load range and the zero page both to be identity-mapped at entry, so that `startup_64` can dereference `RSI` without a translation fault before it builds its own permanent page tables. Because 1 GiB of coverage includes `0x7000`, `0x10_0000`, `0x2_0000`, and the initial stack at `0x8FF0`, the three-level map is sufficient for boot.

Firecracker builds these tables in `setup_page_tables()` in `src/vmm/src/arch/x86_64/regs.rs`. The kernel's first task after `startup_64` validates `boot_params` is to build its own permanent page tables, at which point the VMM's boot tables are no longer referenced.

FPU State

`KVM_SET_FPU` initializes floating-point state to the architectural defaults: `fcw = 0x037F` (x87 control word with all exception masks set and double-extended precision, `PC = 10b`) and `mxcsr = 0x1F80` (MXCSR with all SSE exception masks set and round-to-nearest). The kernel overwrites these during `fpu__init_cpu()`, but KVM requires a valid initial state before entry.

The Full Boot Sequence

With those pieces in place, the boot sequence from `KVM_RUN` to `start_kernel` is a straight line with no firmware detour:

```
sequenceDiagram
    participant VMM as VMM process
    participant KVM as KVM kernel module
    participant CPU as Guest vCPU
    participant K64 as "startup_64 (head_64.S)"
    participant SK as "start_kernel (main.c)"

    VMM->>KVM: KVM_SET_TSS_ADDR, KVM_SET_IDENTITY_MAP_ADDR
    VMM->>KVM: KVM_CREATE_VCPU
    VMM->>VMM: load ELF segments at 0x10_0000
    VMM->>VMM: write boot_params at 0x7000
    VMM->>VMM: write cmdline at 0x2_0000
    VMM->>VMM: write initrd at top of DRAM
    VMM->>VMM: write GDT/IDT at 0x500/0x520
    VMM->>VMM: write page tables at 0x9000–0xB000
    VMM->>KVM: KVM_SET_REGS (RIP=e_entry, RSI=0x7000, ...)
    VMM->>KVM: KVM_SET_SREGS (CR0, CR3, CR4, EFER, GDT, ...)
    VMM->>KVM: KVM_SET_FPU
    VMM->>KVM: KVM_RUN
    KVM->>CPU: VMLAUNCH (Intel VMX) / VMRUN (AMD SVM)
    CPU->>K64: first instruction at RIP
    Note over K64: reads RSI → boot_params<br/>validates magic numbers<br/>reads
e820_table, cmd_line_ptr
    K64->>K64: builds permanent page tables
    K64->>SK: x86_64_start_kernel() → start_kernel()
```

No BIOS runs. No UEFI runs. No GRUB runs. No decompressor runs. The path from `KVM_RUN` to `startup_64` is a single VM-entry, and the path from `startup_64` to `start_kernel` is kernel code running on a machine that the VMM fully configured before handing over control.

The PVH Path

PVH — "Para-Virtualised Hardware," formally the x86/HVM direct boot ABI — offers an alternative that removes the 64-bit paging requirement from the VMM. The kernel's PVH entry point accepts a 32-bit protected-mode state with paging disabled and transitions to long mode itself. That is a simpler entry contract for the VMM, but it means the kernel has more work to do before it can access 64-bit memory.

The ABI is signaled by an ELF PT_NOTE segment in the kernel image, with note name `"Xen"` (four bytes with null terminator) and note type `XEN_ELFNOTE_PHYS32_ENTRY` = 18. The note value is the 32-bit physical address of the PVH entry function in `arch/x86/platform/pvh/enlighten.c`. Linux gained `CONFIG_PVH` in version 5.0; any kernel built with that option exposes this entry point to any hypervisor, not just Xen. Firecracker merged PVH support in release v1.12.0 (PR #5048).

The data structure at entry is `hvm_start_info`, placed at guest-physical address `PVH_INFO_START = 0x6000` in Firecracker's layout:

```
#define XEN_HVM_START_MAGIC_VALUE 0x336ec578

struct hvm_start_info {
    uint32_t magic;          /* must == 0x336ec578         */
    uint32_t version;        /* 0 = v0, 1 = v1             */
    uint32_t flags;          /* SIF_xxx flags               */
    uint32_t nr_modules;     /* count of modules            */
    uint64_t modlist_paddr;  /* phys addr of hvm_modlist_entry[] */
    uint64_t cmdline_paddr;  /* phys addr of command line    */
    uint64_t rsdp_paddr;     /* phys addr of ACPI RSDP      */
    /* v1 additions: */
    uint64_t memmap_paddr;   /* phys addr of memory map      */
    uint32_t memmap_entries; /* entry count (0 = no map)     */
    uint32_t reserved;       /* must be zero                */
};
```

The magic `0x336ec578` is the ASCII string "xEn3" with the high bit of 'E' set — a Xen convention that predates the ABI's hypervisor-agnostic rebranding. The initrd rides in an `hvm_modlist_entry` (32 bytes: `paddr u64`, `size u64`, `cmdline_paddr u64`, `reserved u64`) pointed to by `modlist_paddr`.

The CPU state the PVH ABI mandates differs from the Linux 64-bit protocol in two critical ways: there is no paging (`CR0.PG = 0`), and the pointer to the info struct goes in `EBX` rather than `RSI`. The full required state:

Register / State	Required value
CPU mode	32-bit protected mode
`CR0`	`PE` (bit 0) set; `PG` (bit 31) cleared
`CR4`	All bits cleared
`CS`	32-bit read/execute, base 0, limit `0xFFFF_FFFF`
`DS`, `ES`, `SS`	32-bit read/write, base 0, limit `0xFFFF_FFFF`
`TR`	32-bit TSS, base 0, limit `0x67`
`EFLAGS`	VM (bit 17), IF (bit 9), TF (bit 8) all cleared
`EBX`	Guest-physical address of `hvm_start_info`

In Firecracker's PVH path, `RBX = PVH_INFO_START = 0x6000`. The GDT entries use 32-bit descriptors (`0xC09B` for code, `0xC093` for data, limit `0xFFFF_FFFF`) instead of the 64-bit flags the Linux protocol uses. `CR0` has only `PE` set — no `PG`. The kernel's PVH entry function at `arch/x86/platform/pvh/enlighten.c` builds its own page tables and switches to long mode, at which point execution joins the same `head_64.S` path that the 64-bit Linux protocol takes.

The practical advantage of PVH is not primarily the simpler VMM entry state — that difference is a few dozen lines of code either way. It is that PVH is naturally extensible to non-Linux kernels. FreeBSD and NetBSD both implement the HVM ABI. Firecracker's PVH support therefore works with any kernel that carries the `XEN_ELFNOTE_PHYS32_ENTRY` note, regardless of whether it is Linux.

What Gets Eliminated

A conventional VM boots through at minimum: BIOS POST, PCI device enumeration, an option ROM for each device, a boot block read from a disk, a second-stage bootloader such as GRUB, a kernel image decompressed in guest memory, and then kernel initialization. Each layer was designed for a world where the boot configuration was unknown at build time and had to be discovered at runtime from the hardware. A microVM VMM knows the configuration statically. The kernel image is pinned at deployment time. The memory layout is fixed. The hardware is synthetic and fully controlled. There is nothing to discover.

Sources And Further Reading

- Linux x86 boot protocol specification (magic numbers, field offsets, 64-bit entry, `setup_data`, protocol version table): https://www.kernel.org/doc/html/latest/arch/x86/boot.html
- Zero-page offset table: https://docs.kernel.org/next/x86/zero-page.html
- `struct boot_params`, `struct setup_header`, `boot_e820_entry`, E820 type constants: https://github.com/torvalds/linux/blob/master/arch/x86/include/uapi/asm/bootparam.h
- `boot_flag: .word 0xAA55`, `header: .ascii "HdrS"`, `version: .word 0x020f`: https://github.com/torvalds/linux/blob/master/arch/x86/boot/header.S
- PVH/HVM direct boot ABI specification (CPU state at entry, CR0 requirement, `EBX = hvm_start_info`): https://xenbits.xen.org/docs/unstable/misc/pvh.html
- `XEN_ELFNOTE_PHYS32_ENTRY = 18`, ELF note name "Xen": https://xenbits.xen.org/docs/unstable/hypercall/x86_64/include,public,elfnote.h.html
- `hvm_start_info` struct layout, magic `0x336ec578`, `hvm_modlist_entry`: https://xenbits.xen.org/docs/unstable/hypercall/x86_64/include,public,arch-x86,hvm,start_info.h.html
- Firecracker kernel format requirements (`vmlinux` ELF on x86_64, `Image` on aarch64): https://github.com/firecracker-microvm/firecracker/blob/main/docs/rootfs-and-kernel-setup.md
- Firecracker v1.12.0 PVH boot (PR #5048): https://github.com/firecracker-microvm/firecracker/blob/main/CHANGELOG.md
- rust-vmm `linux-loader` crate — three formats (ELF, bzImage, PE); fields returned per format: https://github.com/rust-vmm/linux-loader/blob/main/README.md
- `linux-loader` design — ELF entry point, `setup_header` extraction, `init_size` usage: https://github.com/rust-vmm/linux-loader/blob/main/DESIGN.md

- `CONFIG_PVH` first appeared in Linux 5.0: https://www.kernelconfig.io/config_pvh
- QEMU PVH boot timing (~10 ms vs. ~78 ms for compressed): https://stefano-garzarella.github.io/posts/2019-08-23-qemu-linux-kernel-pvh/
- `KVM_SET_TSS_ADDR` and `KVM_SET_IDENTITY_MAP_ADDR` requirements: https://docs.kernel.org/virt/kvm/api.html

Chapter 11: virtio — The Paravirtualized Device Model

Every device the guest needs — a network card, a disk, an entropy source — has to go through the VMM. The question is how. The naive answer is emulation: the VMM impersonates a real piece of hardware, the guest drives it with an unmodified driver, and the VMM translates guest I/O port writes and MMIO accesses into host operations. QEMU's emulation of the 82093AA I/OAPIC, the Intel 82576 Gigabit Ethernet controller, and a dozen other real chips is how most full VMs run. It is also slow, not because the emulation itself is expensive, but because the *interface* was designed for a device that has its own DMA engine, its own FIFO, and a long latency to silicon. The guest issues dozens of register writes to queue one operation, each of which exits the CPU into the VMM and back. The overhead is not in the emulated logic; it is in the round-trip count.

Paravirtualization cuts the round-trip count by giving up the pretense of real hardware. The guest runs a driver that *knows* it is talking to a VMM, and the protocol between them is designed for that context: large batches, a shared memory ring for communication, and a single notification per batch rather than one per operation. The physical-device illusion disappears; in its place is an explicit contract between the driver and the device backend.

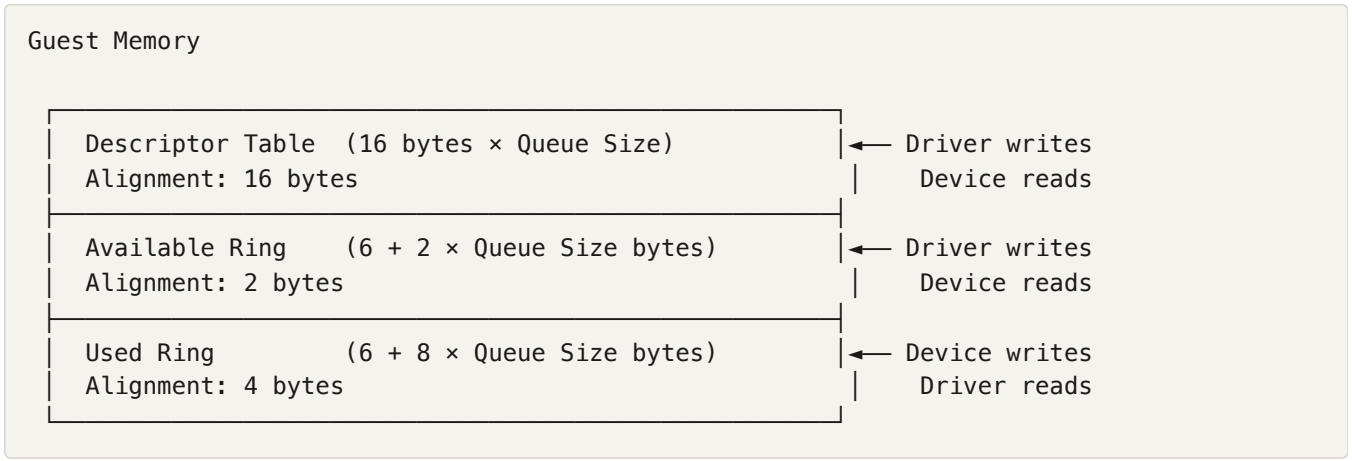
virtio is that contract, standardized. The OASIS virtio Committee Specification v1.2 CS01 (published 1 July 2022) defines the shared-memory ring format, the feature negotiation handshake, the two transports (MMIO and PCI), and the wire protocol for each device class. It is the interface that Firecracker, crosvm, Cloud Hypervisor, and QEMU all implement — which means a Linux guest compiled once can run on any of them without modification, because the driver it loads is the kernel's standard virtio driver, not a VMM-specific one.

The Virtqueue

Every virtio device exposes one or more **virtqueues**: shared-memory rings through which the driver (the guest's kernel driver) submits work and the device (the VMM backend) returns completions. The virtio v1.2 spec defines two queue formats. The **split virtqueue** (spec section 2.7) uses three separate memory regions. The **packed virtqueue** (spec section 2.8), introduced in v1.1 and enabled by feature bit `VIRTIO_F_RING_PACKED = 34`, collapses those three regions into one circular ring plus two small event-suppression structures. Firecracker implements split virtqueues only; the packed format is not supported.

Three Rings, Three Owners

A split virtqueue consists of three physically independent memory regions, each owned exclusively by one side:



The alignment constants are defined in `linux/include/uapi/linux/virtio_ring.h` as `VRING_DESC_ALIGN_SIZE = 16`, `VRING_AVAIL_ALIGN_SIZE = 2`, and `VRING_USED_ALIGN_SIZE = 4`. Queue Size must be a power of two, at least 1, and at most 32,768 (0x8000). Firecracker caps every queue at 256 entries (`FIRECRACKER_MAX_QUEUE_SIZE = 256`).

The ownership rule is strict: the driver never writes to the used ring; the device never writes to the available ring or the descriptor table. This means accesses never race between writer and reader on the same memory. There is still a concurrency hazard — the guest and the VMM run concurrently — but it is bounded to the index fields, which the spec addresses with explicit memory barrier requirements.

The Descriptor Table

Each entry in the descriptor table is a `struct virtq_desc` (16 bytes, all fields in little-endian):

```
struct virtq_desc {
    le64 addr;   /* offset  0: guest-physical buffer address */
    le32 len;    /* offset  8: buffer length in bytes */
    le16 flags;  /* offset 12: control flags */
    le16 next;   /* offset 14: index of next descriptor (if chaining) */
};
```

Three flag bits control how the descriptor is used. `VIRTQ_DESC_F_NEXT = 0x1` means the descriptor is not the last in a chain — the `next` field holds the index of the next descriptor. `VIRTQ_DESC_F_WRITE = 0x2` marks the buffer as device-writable; without it the buffer is device-readable. `VIRTQ_DESC_F_INDIRECT = 0x4` signals that `addr` and `len` point not to data but to an in-memory table of further `virtq_desc` entries, enabled by feature bit `VIRTIO_F_RING_INDIRECT_DESC = 28`. Within an indirect table, only `VIRTQ_DESC_F_WRITE` and `VIRTQ_DESC_F_NEXT` are valid; `VIRTQ_DESC_F_INDIRECT` is forbidden in indirect entries, and the device must ignore `VIRTQ_DESC_F_WRITE` on the outer descriptor that points to the table.

Descriptors chain together to describe a single I/O request. A virtio-blk read, for example, uses three descriptors in a chain: a device-readable header (16 bytes: request type, reserved padding, sector number), one or more device-writable data buffers, and a device-writable one-byte status field. All device-

readable descriptors precede all device-writable ones in the chain — this is a hard split-virtqueue rule, not a convention.

The driver builds these chains by filling descriptor table entries, then publishes the chain by placing the head descriptor's index into the available ring.

The Available Ring

The available ring is the driver's outbox. Its layout (from spec section 2.7.6):

```
struct virtq_avail {
    le16 flags;                  /* VIRTQ_AVAIL_F_NO_INTERRUPT = 1 */
    le16 idx;                    /* where driver will write next head index */
    le16 ring[/* Queue Size */];  /* head indices of published chains */
    le16 used_event;             /* only if VIRTIO_F_EVENT_IDX negotiated */
};
```

The `idx` field wraps naturally at 2^16. The driver increments it by the number of chains it publishes, stores the head indices in `ring[idx % QueueSize]` through `ring[(idx + n - 1) % QueueSize]`, then issues a write memory barrier before notifying the device. The device reads `ring[(last_seen_idx % QueueSize)]` through `ring[(avail.idx - 1) % QueueSize]` to collect new chains.

Notice that `idx` is never reset — it grows monotonically, modulo 2^16. A device that tracks the last `idx` it saw can detect new work without any locking; the index is the only synchronization signal.

The Used Ring

The used ring is the device's completion outbox. Its layout (section 2.7.8):

Each `virtq_used_elem` is 8 bytes. When the device finishes a chain, it writes the head index and byte count into the current used slot, increments `idx`, and — unless notification suppression says otherwise — signals the guest interrupt. The driver scans from its last-seen `idx` to `used.idx - 1` to harvest completions.

Notification Suppression

Left to themselves, driver and device fire an interrupt or a doorbell write after every descriptor batch. For high-throughput paths, that overhead adds up. virtio provides two suppression mechanisms.

The coarse mechanism uses the binary flags: the driver sets `avail.flags = VIRTQ_AVAIL_F_NO_INTERRUPT` to suppress device-to-driver interrupts; the device sets `used.flags = VIRTQ_USED_F_NO_NOTIFY` to suppress driver-to-device kicks. Either side can assert its flag at any time. The tradeoff is crude — all notifications or none.

The fine-grained mechanism, enabled by `VIRTIO_F_RING_EVENT_IDX = 29`, uses threshold fields instead of binary flags. The driver places a target `idx` value into `avail.used_event`; the device fires an interrupt only when `used.idx` reaches that value. The device places a target into `used.avail_event`; the driver kicks only when `avail.idx` reaches it. This lets either side defer a notification precisely until the peer has enough work queued to justify waking up. Firecracker implements the `EVENT_IDX` path and validates the notification-suppression logic and 16-bit index wraparound with Kani formal proofs.

The `virtio-queue` Crate

Firecracker's virtqueue implementation lives in the rust-vmm `virtio-queue` crate (published at `https://crates.io/crates/virtio-queue`). The crate provides two queue types: `Queue` for single-threaded use and `QueueSync` (`Arc<Mutex<Queue>>`) for shared access, both implementing the `QueueT` trait. Key methods include `set_desc_table_address`, `set_avail_ring_address`, `set_used_ring_address`, `set_size`, `set_ready`, `set_event_idx`, `is_valid`, `add_used`, `needs_notification`, `disable_notification`, and `enable_notification`. `AvailIter` is a consuming iterator over available descriptor chain heads; `DescriptorChain` with `DescriptorChainRwIter` separates readable from writable segments cleanly.

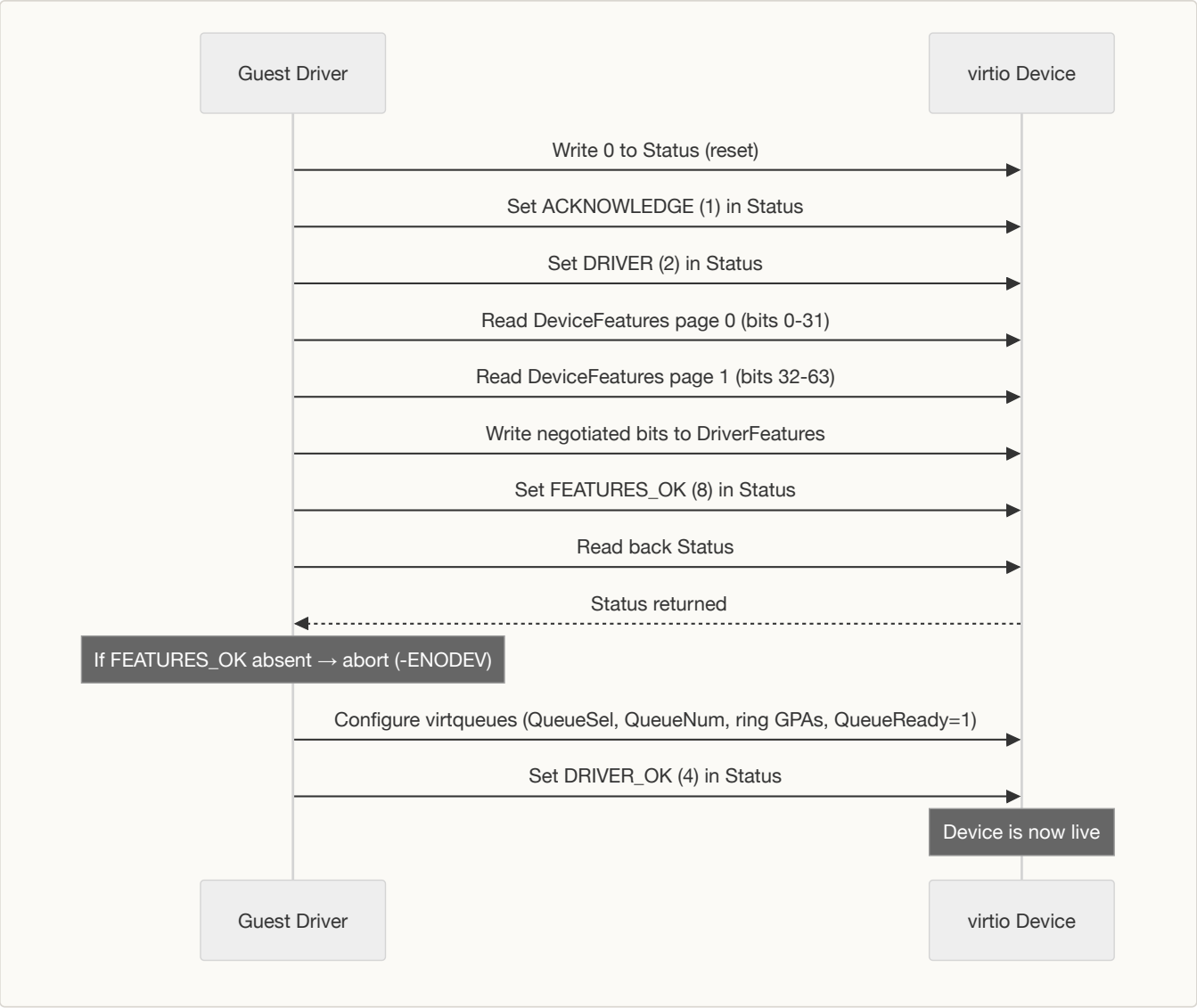
The crate uses Rust `read_volatile` and `write_volatile` with explicit acquire/release memory fences for every ring access, matching the spec's barrier requirements without relying on the compiler to infer them. A Time-To-Live counter limits chain traversal depth to prevent infinite loops from a malicious guest that crafts a circular chain. Used-ring notifications are batched via `prepare_kick()` rather than checked after each `add_used()` call — the crate documents this as a deliberate deviation from spec section 2.6.7.2. The crate targets the virtio v1.1 split virtqueue spec.

Feature Negotiation

The spec imposes a strict handshake before the device becomes usable. This is the mechanism by which a driver compiled three years ago negotiates with a device model compiled last week: each side publishes what it supports; the intersection is what they use. Neither side assumes the other is current.

The Nine-Step Sequence

Spec section 3.1.1 defines the mandatory initialization sequence. The driver must follow these steps in order:



Features are read and written in two 32-bit pages via `DeviceFeaturesSel` and `DriverFeaturesSel`: page 0 covers bits 0–31, page 1 covers bits 32–63. This matters in practice because `VIRTIO_F_VERSION_1 = 32` sits at bit 0 of page 1. A device presenting itself as modern must advertise this bit; a driver that does not acknowledge it is treated as a legacy driver, and a v2 MMIO device must reject initialization if the driver fails to acknowledge it.

The six **device status register** bits (from `linux/include/uapi/linux/virtio_config.h`) are the handshake signals:

Constant	Value	Meaning
`VIRTIO_CONFIG_S_ACKNOWLEDGE`	1	Driver found the device
`VIRTIO_CONFIG_S_DRIVER`	2	Driver knows how to drive it
`VIRTIO_CONFIG_S_FEATURES_OK`	8	Feature negotiation complete
`VIRTIO_CONFIG_S_DRIVER_OK`	4	Driver is live
`VIRTIO_CONFIG_S_NEEDS_RESET`	64	Device needs reset (unrecoverable)
`VIRTIO_CONFIG_S_FAILED`	128	Fatal error

Status starts at 0. The driver must not clear individual bits; only writing 0 resets the register and the device.

Kernel Implementation

`virtio_dev_probe()` in `drivers/virtio/virtio.c` implements steps 2–7: it sets `DRIVER`, calls `virtio_get_features()`, ANDs the device and driver feature tables, calls `dev->config->finalize_features()`, sets `FEATURES_OK`, and reads back status. If `FEATURES_OK` is absent, it returns `-ENODEV`. `virtio_device_ready()` sets `DRIVER_OK` after queue setup completes. `virtio_features_ok()` in `drivers/virtio/virtio.c` checks that `VIRTIO_F_VERSION_1` is in the negotiated set before writing `DriverFeatures` to a modern device.

The Transport-Layer Feature Bits

Most of these bits live in the range `VIRTIO_TRANSPORT_F_START = 28` through `VIRTIO_TRANSPORT_F_END = 42` and apply to every device type. `VIRTIO_F_ANY_LAYOUT` is listed here for completeness — it predates the formal transport range and sits at bit 27, just outside it.

The Linux kernel uapi headers (`virtio_ring.h`) name the indirect-descriptor and event-index bits `VIRTIO_RING_F_INDIRECT_DESC` and `VIRTIO_RING_F_EVENT_IDX`; the OASIS spec uses `VIRTIO_F_RING_INDIRECT_DESC` and `VIRTIO_F_RING_EVENT_IDX` for the same bits (28 and 29). This chapter follows the spec naming.

Constant (OASIS spec)	Bit	Meaning
`VIRTIO_F_ANY_LAYOUT`	27	Device handles any descriptor ordering (predates transport range)
`VIRTIO_F_RING_INDIRECT_DESC`	28	Indirect descriptor tables
`VIRTIO_F_RING_EVENT_IDX`	29	Descriptor-granularity notification suppression
`VIRTIO_F_VERSION_1`	32	Modern device (mandatory for modern devices)
`VIRTIO_F_ACCESS_PLATFORM`	33	IOMMU DMA required
`VIRTIO_F_RING_PACKED`	34	Packed virtqueue format
`VIRTIO_F_IN_ORDER`	35	Buffers used in availability order
`VIRTIO_F_RING_RESET`	40	Per-queue reset

Firecracker advertises `VIRTIO_F_VERSION_1` and `VIRTIO_F_RING_EVENT_IDX` on all its devices. `VIRTIO_F_RING_PACKED` is never advertised because Firecracker does not implement packed virtqueues.

Config Space Atomicity

Device-specific configuration fields (capacity, MAC address, queue pair count, and so on) live in a config space region that can be updated at any time — for example, a network link-state change arriving mid-probe. Spec section 2.5 requires the driver to re-read config fields in a compare-and-retry loop using the `config_generation` field (MMIO offset `0x0fc`) whenever a concurrent change is suspected. The device increments `config_generation` before and after each config update; if the driver reads a different value at the end of a read sequence than at the beginning, it retries.

The MMIO Transport

The MMIO transport (spec section 4.2) exposes the device as a flat register window mapped into the guest's physical address space. There is no bus, no enumeration protocol, no capability list — just a base address and an IRQ number that the VMM communicates to the guest out-of-band.

Register Map

All registers are 4 bytes wide, 4-byte-aligned, at fixed offsets from the base address (from `linux/include/uapi/linux/virtio_mmio.h`):

Register	Offset	Dir	Purpose
`MagicValue`	`0x000`	RO	Must read `0x74726976` ("virt" in LE ASCII)
`Version`	`0x004`	RO	2 = modern; 1 = legacy
`DeviceID`	`0x008`	RO	virtio device type
`VendorID`	`0x00c`	RO	Vendor identifier
`DeviceFeatures`	`0x010`	RO	32-bit feature page
`DeviceFeaturesSel`	`0x014`	WO	Feature page selector (0 or 1)
`DriverFeatures`	`0x020`	WO	Accepted feature bits
`DriverFeaturesSel`	`0x024`	WO	Driver feature page selector
`QueueSel`	`0x030`	WO	Select active queue (0-indexed)
`QueueNumMax`	`0x034`	RO	Maximum queue size
`QueueNum`	`0x038`	WO	Actual queue size (driver chooses)
`QueueReady`	`0x044`	RW	Write 1 to activate queue
`QueueNotify`	`0x050`	WO	Write queue index to kick device
`InterruptStatus`	`0x060`	RO	Bit 0 = used-buffer; bit 1 = config change
`InterruptACK`	`0x064`	WO	Acknowledge interrupt bits
`Status`	`0x070`	RW	Device status register
`QueueDescLow`	`0x080`	WO	Descriptor Table GPA bits 31:0
`QueueDescHigh`	`0x084`	WO	Descriptor Table GPA bits 63:32
`QueueAvailLow`	`0x090`	WO	Available Ring GPA bits 31:0
`QueueAvailHigh`	`0x094`	WO	Available Ring GPA bits 63:32
`QueueUsedLow`	`0x0a0`	WO	Used Ring GPA bits 31:0
`QueueUsedHigh`	`0x0a4`	WO	Used Ring GPA bits 63:32
`ConfigGeneration`	`0x0fc`	RO	Config space atomicity counter
`Config`	`0x100+`	RW	Device-specific config (up to `0xfff`)

The legacy (Version 1) layout adds `GuestPageSize` at `0x028`, `QueueAlign` at `0x03c`, and `QueuePFN` at `0x040`, and collapses the split 64-bit address pairs into a single page-frame number. Modern drivers do not touch these.

Device Discovery

MMIO has no self-describing discovery mechanism (spec section 4.2.1). The guest must learn each device's base address and IRQ from the VMM. Linux's `drivers/virtio/virtio_mmio.c` driver supports three paths: a device tree node with `compatible = "virtio,mmio"`, a kernel command-line parameter `virtio_mmio.device=<size>@<baseaddr>:<irq>[:<id>]` (requires `CONFIG_VIRTIO_MMIO_CMDLINE_DEVICES`), and static platform device registration in board code.

Firecracker uses the command-line path: it appends one `virtio_mmio.device=...` entry per device to the kernel command line at boot, advertising each device's MMIO window size, base address, and IRQ. An open issue (#2519) proposes replacing this with a device tree blob passed via the `setup_data` boot protocol field, but as of this writing the issue is not merged.

Firecracker's MMIO Backend

Firecracker's MMIO transport is implemented in `src/vmm/src/devices/virtio/transport/mmio.rs` as `MmioTransport`, which implements the `BusDevice` trait. Guest writes to MMIO space cause VM exits; the VMM dispatches them through `BusDevice::read` and `BusDevice::write`.

A few Firecracker-specific constants are worth naming. `MMIO_VERSION = 2` is hardcoded — the device always presents as modern. `VENDOR_ID = 0` deviates from the spec's recommended value of `0x1AF4` (Red Hat, Inc.) and mirrors the crosvm convention. Reading `DeviceFeatures` with `DeviceFeaturesSel = 1` ORs in `0x1` unconditionally, so `VIRTIO_F_VERSION_1` (bit 32) is always visible to the driver regardless of what the inner device model advertises. `set_device_status()` enforces the spec state machine with a `VALID_TRANSITIONS` table; any status write that is not a legal transition logs a warning. Transition to `DRIVER_OK` calls `locked_device().activate()`, which hands control to the device backend — at that point, the virtqueues are live and the device can begin processing descriptors.

The guest kernel requires `CONFIG_VIRTIO_MMIO=y` and `CONFIG_VIRTIO_MMIO_CMDLINE_DEVICES=y` to use Firecracker MMIO devices.

The PCI Transport

The PCI transport (spec section 4.1) is self-describing. The guest scans the PCI bus, finds devices with Vendor ID `0x1AF4` (Red Hat, Inc.), and walks each device's PCI capability list to find the five vendor-specific capability structures that virtio-PCI defines. No out-of-band communication is needed: the bus itself tells the driver where everything is.

Device IDs

PCI device IDs split into two ranges. Legacy (transitional) devices use IDs `0x1000` – `0x103F`; modern devices use `0x1040` + the virtio device ID, so virtio-net is `0x1041`, virtio-blk is `0x1042`, virtio-rng is `0x1044`, and virtio-vsock is `0x1053`.

Five Capability Structures

Each capability uses `cap_vndr = PCI_CAP_ID_VNDR` (identifying it as a vendor-specific capability) and a `cfg_type` field that says which of the five roles it plays (from `linux/include/uapi/linux/virtio_pci.h`):

`cfg_type`	Value	Purpose
`VIRTIO_PCI_CAP_COMMON_CFG`	1	Common configuration struct (`virtio_pci_common_cfg`)
`VIRTIO_PCI_CAP_NOTIFY_CFG`	2	Queue doorbell addresses
`VIRTIO_PCI_CAP_ISR_CFG`	3	Interrupt status byte
`VIRTIO_PCI_CAP_DEVICE_CFG`	4	Device-specific configuration
`VIRTIO_PCI_CAP_PCI_CFG`	5	Alternative PCI config-space access window

`struct virtio_pci_cap` records which BAR holds the region (`bar` , 0–5), the byte offset within that BAR (`offset`), and the region's length (`length`). The notification capability also carries `notify_off_multiplier` ; the doorbell address for queue N is `cap.offset + queue_notify_off × notify_off_multiplier` .

`struct virtio_pci_common_cfg` exposes the feature selectors and data fields, the queue count, the device status register, `config_generation` , the queue selector and size, `queue_enable` , and the split 64-bit GPA fields `queue_desc_lo/hi` , `queue_avail_lo/hi` , and `queue_used_lo/hi` — a superset of the MMIO register map, accessed through a memory-mapped struct rather than individual register offsets.

Why Firecracker Originally Chose MMIO

PCI bus enumeration, ACPI table parsing, and MSI/MSI-X interrupt wiring each add work to the guest boot path. For Firecracker's original target — the serverless VM that must start in under 150 ms — those milliseconds matter. MMIO requires none of that infrastructure, the device model is simpler, and the command-line discovery mechanism is a handful of string appends.

PCI transport was later added to Firecracker behind `--enable-pci` . Benchmarks from the Firecracker team (discussion #4845) show the tradeoff concretely: block synchronous reads improve by about 50%, block synchronous writes by 46% (on a 1-vCPU VM), network transmit throughput by 2–11%, and network receive throughput by 9–17%. Latency drops roughly 27%. The cost is an approximately 8% slower boot on VMs under 4 GiB. The fundamental reason is interrupt delivery: MMIO uses level-triggered interrupts that require a VM exit per notification; MSI-X can deliver interrupts without a VMM-side exit. PCI is the right answer when throughput dominates; MMIO is the right answer when boot time does.

Enabling PCI mode requires additional guest kernel configuration: `CONFIG_PCI`, `CONFIG_PCI_MMCONFIG`, `CONFIG_PCI_MSI`, `CONFIG_PCIEPORTBUS`, `CONFIG_VIRTIO_PCI`, `CONFIG_BLK_MQ_PCI`, `CONFIG_PCI_HOST_COMMON`, and `CONFIG_PCI_HOST_GENERIC`. The guest must not pass `pci=off` on its command line.

The Devices That Matter

The five device types Firecracker exposes cover everything a modern serverless workload needs: a network path, a block device, a host-guest socket channel, memory pressure signaling, and entropy. Each is a separate protocol layered on top of the virtqueue machinery.

virtio-net (Device ID 1)

The network device presents the guest with an Ethernet interface backed by a TAP device on the host.

TAP setup. Firecracker opens `/dev/net/tun` with `O_RDWR | O_NONBLOCK | O_CLOEXEC` and calls `TUNSETIFF` (`_IOW('T', 202, int)`) with three flags: `IFF_TAP = 0x0002` (Ethernet frames, not raw IP), `IFF_NO_PI = 0x1000` (do not prepend the four-byte `struct tun_pi` packet info header), and `IFF_VNET_HDR = 0x4000` (prepend or strip a `virtio_net_hdr` on each frame). Two more ioctls complete the setup: `TUNSETOFFLOAD` (`_IOW('T', 208, unsigned int)`) advertises which checksum offloads the tap device can handle, and `TUNSETVNETHDRSZ` (`_IOW('T', 216, int)`) tells the kernel to use the 12-byte `virtio_net_hdr_v1` format rather than the legacy 10-byte form. Interface name is a 16-byte array matching `IFNAMSIZ`. These constants and structures are defined in `linux/include/uapi/linux/if_tun.h`.

> *Before running commands that open `/dev/net/tun` or manage TAP devices, the process needs either `CAP_NET_ADMIN` or a pre-created TAP interface. On a production host Firecracker relies on the jailer to set up the TAP before dropping privileges; on a development machine, `sudo ip tuntap add dev tap0 mode tap` creates one manually.*

The virtio-net header. Every frame crossing the TAP/virtqueue boundary carries a `virtio_net_hdr_v1` (12 bytes) that describes the offload state of the packet (defined in `linux/include/uapi/linux/virtio_net.h`):

Offset	Field	Notes
0	`flags`	`VIRTIO_NET_HDR_F_NEEDS_CSUM = 1`
1	`gso_type`	NONE=0, TCPV4=1, UDP=3, TCPV6=4, ECN flag=0x80
2–3	`hdr_len`	Total L2+L3+L4 header length
4–5	`gso_size`	Desired MSS for segmentation
6–7	`csum_start`	Byte offset where checksum computation begins
8–9	`csum_offset`	Offset from `csum_start` to place the checksum
10–11	`num_buffers`	Merged receive buffer count (if `VIRTIO_NET_F_MRG_RXBUF`)

Queues. Firecracker implements exactly two virtqueues: `RX_INDEX = 0` and `TX_INDEX = 1`, each capped at `NET_QUEUE_MAX_SIZE = 256` descriptors. The spec allows a multi-queue extension (`VIRTIO_NET_F_MQ = 22`) with one transmit and one receive queue per CPU, but Firecracker does not implement it; each virtio-net device has a single queue pair. `MAX_BUFFER_SIZE = 65562` bytes (64 KiB plus the virtio-net header overhead) is the largest receive buffer the device will accept.

Feature bits Firecracker advertises. From `linux/include/uapi/linux/virtio_net.h`, Firecracker sets: `VIRTIO_NET_F_CSUM (0)`, `VIRTIO_NET_F_GUEST_CSUM (1)`, `VIRTIO_NET_F_GUEST_TSO4 (7)`, `VIRTIO_NET_F_GUEST_TSO6 (8)`, `VIRTIO_NET_F_GUEST_UFO (10)`, `VIRTIO_NET_F_HOST_TSO4 (11)`, `VIRTIO_NET_F_HOST_TSO6 (12)`, `VIRTIO_NET_F_HOST_UFO (14)`, `VIRTIO_NET_F_MRG_RXBUF (15)`, plus `VIRTIO_F_RING_EVENT_IDX (29)` and `VIRTIO_F_VERSION_1 (32)`. `VIRTIO_NET_F_MAC (5)` is added when a MAC address is configured; `VIRTIO_NET_F_MTU (3)` when an MTU override is set.

The config space struct (`virtio_net_config`) carries MAC (6 bytes), status (2 bytes), `max_virtqueue_pairs` (2 bytes), MTU (2 bytes), speed in Mbps (4 bytes), and duplex (1 byte).

virtio-blk (Device ID 2)

The block device exposes a single virtqueue to the guest. Firecracker implements `BLOCK_NUM_QUEUES = 1`, sized to 256 descriptors. `IO_URING_NUM_ENTRIES = 128` (half the queue depth) because one block request typically spans two to three descriptors; a full 256-entry submission ring would overflow an io_uring ring of the same size.

The sector model. `SECTOR_SIZE = 512` bytes (1 << 9). The `capacity` field in the config struct is a `u64` reporting the total sector count. A guest trying to read sector N at offset `N × 512` from the start of the backing file or device.

Request layout. Each I/O request is a three-descriptor chain:

1. A 16-byte device-readable header: `type` (u32), `reserved` (u32), `sector` (u64).

2. One or more data buffers — device-readable for writes, device-writable for reads.
3. A one-byte device-writable status field: `VIRTIO_BLK_S_OK = 0`, `VIRTIO_BLK_S_IOERR = 1`, or `VIRTIO_BLK_S_UNSUPP = 2`.

The `type` field in the header selects the operation: `VIRTIO_BLK_T_IN = 0` (read), `VIRTIO_BLK_T_OUT = 1` (write), `VIRTIO_BLK_T_FLUSH = 4` (cache flush), `VIRTIO_BLK_T_GET_ID = 8` (identify device: returns a 20-byte ASCII string). All defined in `linux/include/uapi/linux/virtio_blk.h`.

Feature bits Firecracker advertises. `VIRTIO_F_VERSION_1 (32)` and `VIRTIO_F_RING_EVENT_IDX (29)` always. `VIRTIO_BLK_F_FLUSH (9)` when the backing disk is in writeback-cache mode. `VIRTIO_BLK_F_RO (5)` when the disk is read-only.

virtio-vsock (Device ID 19)

vsock gives the guest and host a socket channel without a network interface. The guest opens a socket with `socket(AF_VSOCK, SOCK_STREAM, 0)` and addresses the host by its well-known CID. This is the channel Firecracker uses for its API proxy feature and for guest agent communication in richer microVM platforms.

The address family `AF_VSOCK` was introduced in Linux 4.8. Each endpoint is addressed by a (CID, port) pair. Reserved CIDs: `VMADDR_CID_HYPERVISOR = 0`, `VMADDR_CID_LOCAL = 1`, `VMADDR_CID_HOST = 2`, `VMADDR_CID_ANY = 0xFFFFFFFF`. Firecracker sets `VSOCK_HOST_CID = 2` for the host-side endpoint.

Queues. Firecracker implements three queues (`VSOCK_NUM_QUEUES = 3`): RXQ (index 0) for data from host to guest, TXQ (index 1) for data from guest to host, and EVQ (index 2) for event messages. All three are sized to 256 descriptors. Each descriptor chain encodes exactly one vsock packet: a 44-byte header followed by an optional payload up to `MAX_PKT_BUF_SIZE = 65536` bytes.

The header. `virtio_vsock_hdr` is 44 bytes, packed:

Offset	Field	Type	Notes
0–7	`src_cid`	le64	Source context ID
8–15	`dst_cid`	le64	Destination context ID
16–19	`src_port`	le32	Source port
20–23	`dst_port`	le32	Destination port
24–27	`len`	le32	Payload byte count
28–29	`type`	le16	Socket type (1=STREAM, 2=SEQPACKET)
30–31	`op`	le16	Operation code
32–35	`flags`	le32	Operation-specific flags
36–39	`buf_alloc`	le32	Receiver buffer allocation (flow control)
40–43	`fwd_cnt`	le32	Bytes consumed by receiver (flow control)

The `op` field drives the connection state machine. `VIRTIO_VSOCK_OP_REQUEST = 1` initiates a connection; `VIRTIO_VSOCK_OP_RESPONSE = 2` accepts it; `VIRTIO_VSOCK_OP_RST = 3` rejects or aborts; `VIRTIO_VSOCK_OP_SHUTDOWN = 4` begins a graceful close; `VIRTIO_VSOCK_OP_RW = 5` carries data; `VIRTIO_VSOCK_OP_CREDIT_UPDATE = 6` and `VIRTIO_VSOCK_OP_CREDIT_REQUEST = 7` implement receive-window flow control through `buf_alloc` and `fwd_cnt` in the header — the receiver advertises available buffer space, and the sender tracks how much of it it has consumed.

Feature bits. Firecracker advertises `VIRTIO_F_VERSION_1 (32)`, `VIRTIO_F_IN_ORDER (35)`, and `VIRTIO_F_RING_EVENT_IDX (29)`, combined as `AVAIL_FEATURES = (1 << 32) | (1 << 35) | (1 << 29)`. The device-specific feature `VIRTIO_VSOCK_F_SEQPACKET = 1` (SOCK_SEQPACKET support) is not advertised.

virtio-balloon (Device ID 5)

The balloon device lets the host reclaim memory from a running guest without stopping it. The host writes a target page count into the `num_pages` field of the config struct; the guest driver inflates the balloon — pins that many 4 KiB pages, reports them to the host, and the host calls `madvise(MADV_DONTNEED)` on those guest-physical ranges. The guest kernel can no longer access them efficiently; the host OS reclaims the physical pages. When the host writes a smaller target, the guest deflates: it releases the pinned pages back to its own allocator and the host stops MADV_DONTNEED-ing them. Re-access by the guest returns zero-filled pages on demand.

This is the mechanism by which Firecracker supports memory overcommit: a fleet of microVMs can collectively commit more memory than the host has, and the balloon keeps actual physical usage within bounds.

Config struct. `virtio_balloon_config` (from `linux/include/uapi/linux/virtio_balloon.h`):

Field	Type	Meaning
`num_pages`	le32	Host-requested balloon size in 4 KiB pages
`actual`	le32	Current balloon size in 4 KiB pages
`free_page_hint_cmd_id`	le32	Command ID for free page hinting
`poison_val`	le32	Page poison value

All balloon accounting is in 4 KiB pages; 256 pages equals 1 MiB. The host sets `num_pages`; the guest updates `actual` as it completes inflation or deflation.

Queues. The inflate queue (index 0) and deflate queue (index 1) are always present, sized to 128 descriptors in Firecracker. Additional queues appear conditionally: the stats queue (index 2) when `VIRTIO_BALLOON_F_STATS_VQ` is negotiated, a free-page-hinting queue (index 3) when `VIRTIO_BALLOON_F_FREE_PAGE_HINT` is set, and a page-reporting queue (index 4) when `VIRTIO_BALLOON_F_REPORTING` is set.

Feature bits Firecracker advertises. `VIRTIO_F_VERSION_1 (32)`, `VIRTIO_BALLOON_F_DEFLATE_ON_OOM (2)` (the guest automatically deflates if the host OOM killer fires), `VIRTIO_BALLOON_F_STATS_VQ (1)`, `VIRTIO_BALLOON_F_FREE_PAGE_HINT (3)`, and `VIRTIO_BALLOON_F_REPORTING (5)`.

Statistics. The stats queue carries tagged 8-byte entries with a `u16` tag and a `u64` value. The defined tags include swap-in and swap-out counts (`VIRTIO_BALLOON_S_SWAP_IN = 0`, `VIRTIO_BALLOON_S_SWAP_OUT = 1`), page fault counts (`VIRTIO_BALLOON_S_MAJFLT = 2`, `VIRTIO_BALLOON_S_MINFLT = 3`), free and total memory (`VIRTIO_BALLOON_S_MEMFREE = 4`, `VIRTIO_BALLOON_S_MEMTOT = 5`), available memory `(VIRTIO_BALLOON_S_AVAIL = 6)`, page cache size (`VIRTIO_BALLOON_S_CACHES = 7`), and OOM kill count (`VIRTIO_BALLOON_S_OOM_KILL = 10`). The statistics polling interval is configurable in Firecracker; setting it to 0 disables polling. The guest kernel requires `CONFIG_MEMORY_BALLOON=y` and `CONFIG_VIRTIO_BALLOON=y`.

virtio-rng (Device ID 4)

The entropy device is the simplest virtio device in the spec: `linux/include/uapi/linux/virtio_rng.h` contains no device-specific feature bits — it includes only `virtio_ids.h` and `virtio_config.h`. There is no device-specific config space. The entire protocol fits in a paragraph.

Firecracker exposes one queue (`RNG_NUM_QUEUES = 1`) and advertises only `VIRTIO_F_VERSION_1 (32)`. The queue direction is device-writable only: the guest posts write-only descriptors pointing to buffers it wants filled with entropy, and the device fills them. The guest never sends data to the device.

`MAX_ENTROPY_BYTES = 65536` (64 KiB) is the cap on bytes served per request, preventing host memory exhaustion from a malicious guest that crafts overlapping descriptor chains pointing to enormous ranges. Firecracker draws entropy from `aws_lc_rs::rand` (the AWS LibCrypto Rust bindings), not from `/dev/random` or `getrandom()` directly. Rate limiting is available via the Firecracker API, with independent controls for bytes-per-second and operations-per-second.

The simplicity is the point. A random number device has no protocol state, no connection setup, no flow control, and no error conditions beyond buffer exhaustion. It is what virtio looks like when nothing is left to remove.

Wiring It Together

```
flowchart TB
    gk["Guest Kernel Driver"]
    vq["Virtqueue<br/>(desc / avail / used rings)"]
    mmio["virtio-MMIO or PCI Transport"]
    vmm["VMM Device Backend<br/>(Firecracker)"]
    tap["TAP /dev/net/tun"]
    blkfile["Block backing file"]
    vsockunix["Unix socket<br/>(vsock muxer)"]
    awslc["aws_lc_rs::rand"]

    gk -->|"write head idx to avail.ring,\nkick QueueNotify"| vq
    vq -->|"VM exit on register write"| mmio
    mmio -->|"dispatch to activate()d device"| vmm
    vmm -->|"net: read/write virtio_net_hdr + frame"| tap
    vmm -->|"blk: read/write 512-byte sectors"| blkfile
    vmm -->|"vsock: virtio_vsock_hdr + payload"| vsockunix
    vmm -->|"rng: fill entropy bytes"| awslc
    vmm -->|"write id+len to used.ring,\ninterrupt guest"| vq
    vq -->|"driver polls used.idx"| gk
```

Every device Firecracker ships uses the same queue depth (256), the same notification suppression path (`VIRTIO_F_RING_EVENT_IDX`), and the same memory fence discipline (`read_volatile` / `write_volatile` with acquire/release barriers). The per-device protocol differences are entirely in the descriptor chain layout and the config space struct — the queue machinery is shared.

Sources And Further Reading

- OASIS virtio Committee Specification v1.2, CS01, 1 July 2022 (HTML): https://docs.oasis-open.org/virtio/virtio/v1.2/cs01/virtio-v1.2-cs01.html
- OASIS virtio v1.2 CS01 (PDF): https://docs.oasis-open.org/virtio/virtio/v1.2/cs01/virtio-v1.2-cs01.pdf
- `oasis-tcs/virtio-spec` canonical C headers (`virtio-queue.h`): https://github.com/oasis-tcs/virtio-spec/blob/master/virtio-queue.h

- Linux UAPI `linux/virtio_ring.h`: https://github.com/torvalds/linux/blob/master/include/uapi/linux/virtio_ring.h
- Linux UAPI `linux/virtio_config.h`: https://github.com/torvalds/linux/blob/master/include/uapi/linux/virtio_config.h
- Linux UAPI `linux/virtio_mmio.h`: https://github.com/torvalds/linux/blob/master/include/uapi/linux/virtio_mmio.h
- Linux UAPI `linux/virtio_pci.h`: https://github.com/torvalds/linux/blob/master/include/uapi/linux/virtio_pci.h
- Linux UAPI `linux/virtio_ids.h`: https://github.com/torvalds/linux/blob/master/include/uapi/linux/virtio_ids.h
- Linux UAPI `linux/virtio_net.h`: https://github.com/torvalds/linux/blob/master/include/uapi/linux/virtio_net.h
- Linux UAPI `linux/virtio_blk.h`: https://github.com/torvalds/linux/blob/master/include/uapi/linux/virtio_blk.h
- Linux UAPI `linux/virtio_vsock.h`: https://github.com/torvalds/linux/blob/master/include/uapi/linux/virtio_vsock.h
- Linux UAPI `linux/virtio_balloon.h`: https://github.com/torvalds/linux/blob/master/include/uapi/linux/virtio_balloon.h
- Linux UAPI `linux/virtio_rng.h`: https://github.com/torvalds/linux/blob/master/include/uapi/linux/virtio_rng.h
- Linux UAPI `linux/if_tun.h`: https://github.com/torvalds/linux/blob/master/include/uapi/linux/if_tun.h
- Linux `drivers/virtio/virtio.c` (feature negotiation implementation): https://github.com/torvalds/linux/blob/master/drivers/virtio/virtio.c
- Linux `drivers/virtio/virtio_mmio.c`: https://github.com/torvalds/linux/blob/master/drivers/virtio/virtio_mmio.c
- Linux kernel virtio driver API documentation: https://docs.kernel.org/driver-api/virtio/virtio.html
- `vsock(7)` man page: https://man7.org/linux/man-pages/man7/vsock.7.html
- rust-vmm `virtio-queue` crate: https://crates.io/crates/virtio-queue
- rust-vmm `virtio-queue` README: https://github.com/rust-vmm/vm-virtio/blob/main/virtio-queue/README.md
- Firecracker `src/vmm/src/devices/virtio/transport/mmio.rs`: https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/devices/virtio/transport/mmio.rs
- Firecracker `src/vmm/src/devices/virtio/net/device.rs`: https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/devices/virtio/net/device.rs
- Firecracker `src/vmm/src/devices/virtio/net/tap.rs`: https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/devices/virtio/net/tap.rs

- Firecracker `src/vmm/src/devices/virtio/block/virtio/device.rs`: https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/devices/virtio/block/virtio/device.rs
- Firecracker `src/vmm/src/devices/virtio/vsock/device.rs`: https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/devices/virtio/vsock/device.rs
- Firecracker `src/vmm/src/devices/virtio/balloon/device.rs`: https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/devices/virtio/balloon/device.rs
- Firecracker `src/vmm/src/devices/virtio/rng/device.rs`: https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/devices/virtio/rng/device.rs
- Firecracker ballooning documentation: https://github.com/firecracker-microvm/firecracker/blob/main/docs/ballooning.md
- Firecracker network setup documentation: https://github.com/firecracker-microvm/firecracker/blob/main/docs/network-setup.md
- Firecracker issue #2519 (virtio-mmio device tree): https://github.com/firecracker-microvm/firecracker/issues/2519
- Firecracker PCI performance discussion #4845: https://github.com/firecracker-microvm/firecracker/discussions/4845
- Kani formal verification of Firecracker virtio queue: https://model-checking.github.io/kani-verifier-blog/2022/07/13/using-the-kani-rust-verifier-on-a-firecracker-example.html

Chapter 12: The Minimal Machine Model

Every x86_64 virtual machine carries baggage from 1981. The IBM PC used a Zilog 8253 timer, an Intel 8259A interrupt controller, and a National Semiconductor 16550 UART. Later revisions cascaded a second 8259A for more IRQ lines and added an Intel 8042 to manage keyboards. These chips persisted through PC/AT, through the ISA bus, through PCI, through ACPI, and into the UEFI era. Modern Linux kernels probe for them on every x86_64 boot. They are not optional: the kernel's TSC calibration path gates on the presence of a legacy PIC, and the early console driver writes to the UART before any device tree or ACPI table has been parsed.

So the question for a VMM trying to offer the smallest safe attack surface is not "which devices do we want?" but "which devices can we remove without breaking the kernel?" For Firecracker the answer is: no PCI bus, no ACPI power management, no BIOS or firmware — but yes to a 16550A UART at COM1, yes to an i8042 stub that can reset the CPU, and yes to the three interrupt controllers KVM provides in-kernel for free. This chapter traces why those choices are forced, how the VMM routes I/O to the right emulator, and where QEMU's `microvm` machine type arrives at the same destination via a different road.

The Devices the Kernel Still Expects

16550A UART: The Console That Runs Before the Console

Linux's `earlycon` driver is the first piece of kernel code that can write to a human-readable output. It runs before the page allocator, before `printk`'s ring buffer, and before any PCI or USB enumeration. The only hardware it can target is a memory-mapped or I/O-port UART at a fixed, known address. For x86_64, that address is `0x3F8` — COM1, as it has been since the IBM PC.

The kernel boot parameter `earlycon=uart8250,io,0x3f8` tells the early console driver to use PIO mode at that address. The kernel's serial console, activated later by `console=ttyS0,115200`, uses the same register layout. Neither path negotiates the UART's location: they assume COM1, assume GSI 4, and start writing to the THR register at offset 0 from the base address.

Firecracker satisfies this expectation with the `vm-superio` crate (`rust-vmm/vm-superio`), which emulates the 16550A register file. The constants are in `src/vmm/src/device_manager/legacy.rs`: `SERIAL_PORT_ADDRESS = 0x3f8`, `SERIAL_PORT_SIZE = 0x8`, and `COM1_GSI = 4`.

The emulated UART presents itself as a 16550A by setting bits 7:6 of the Interrupt Identification Register (IIR) to `0b11` on every IIR read (`IIR_FIFO_BITS = 0b1100_0000` in `vm-superio/src/serial.rs`). That is exactly the check `serial8250_config_port()` in `8250_port.c` performs — it tests IIR bits 7:6 for `0b11` to classify the device as 16550A-compatible. The internal FIFO is 64 bytes (`FIFO_SIZE =`

`0x40`). The default baud divisor is `DEFAULT_BAUD_DIVISOR_LOW = 0x0C` with `DEFAULT_BAUD_DIVISOR_HIGH = 0x00`: divisor 12 at the 1.8432 MHz base clock gives 9600 bps, though the guest kernel ignores the actual baud rate because the host-side serializer does not depend on it.

The register map is eight bytes wide:

Offset	Name	Direction	Notes
0	RBR / THR	R / W	Receive buffer / transmit holding; DLAB_LOW when LCR bit 7 set
1	IER / DLAB_HIGH	R / W	Interrupt Enable (bits 3:0 valid); divisor high byte when DLAB set
2	IIR / FCR	R / W	IIR on read: bits 7:6 = `11` signals 16550A
3	LCR	R / W	Line Control; bit 7 = DLAB (Divisor Latch Access Bit)
4	MCR	R / W	Modem Control (5 bits)
5	LSR	R	Line Status; bit 5 = THR empty (earlycon polls this before each byte)
6	MSR	R	Modem Status: CTS=0x10, DSR=0x20, RI=0x40, DCD=0x80
7	SCR	R / W	Scratch Register

The `earlycon` path never enables interrupts. It polls LSR bit 5 ("THR empty") before writing each byte to offset 0. The interrupt-driven path — used once the full serial driver loads — sets `IER_RDA_BIT = 0b0000_0001` to be notified of received data and `IER_THR_EMPTY_BIT = 0b0000_0010` to be notified when the transmit holding register drains. Interrupts are delivered via a `Trigger` trait backed by a Linux `eventfd`; KVM translates the eventfd signal into an IRQ injection on GSI 4.

One production wrinkle: Firecracker's design document notes that the serial console is disabled in production builds because it may expose guest data through timing side channels. The emulated register file remains, but guest-side output is not forwarded to the host. Development and debugging images re-enable it via the boot configuration. From Firecracker v1.8.0 onward, the UART is also described in the ACPI DSDT as `_SB_.COM1` with EISA HID `PNP0501`, so an ACPI- aware guest can discover it without a kernel command-line hint.

i8042: A Controller Present Only to Reboot

The Intel 8042 was the PS/2 keyboard controller in every PC/AT. It scanned keys, debounced signals, and shared the CPU IRQ line (GSI 1) with the PS/2 mouse. In a microVM, none of that matters. No human is typing into a Firecracker instance. The i8042 is emulated for one reason: the Linux kernel, when asked to reboot with the kernel parameter `reboot=k`, resets the CPU by writing command `0xFE` to the i8042 command port at `0x64`.

Firecracker's i8042 implementation is in `src/vmm/src/devices/legacy/i8042.rs`. It occupies five bytes starting at `I8042_KDB_DATA_REGISTER_ADDRESS = 0x060`: the data register at `0x60` and the status/command register at offset 4 (`OFS_STATUS = 4`), which maps to `0x64`. IRQ GSI 1 (`KBD_EVT_GSI = 1`) is wired for the keyboard port; the PS/2 mouse (IRQ 12) is not emulated.

The command set is minimal:

Constant	Value	Effect
`CMD_READ_CTR`	`0x20`	Read the control register
`CMD_WRITE_CTR`	`0x60`	Write the control register
`CMD_READ_OUTP`	`0xD0`	Read the output port
`CMD_WRITE_OUTP`	`0xD1`	Write the output port
`CMD_RESET_CPU`	`0xFE`	Signal CPU reset; triggers the `reset_evt` eventfd

When `CMD_RESET_CPU` arrives, Firecracker signals a `reset_evt` eventfd. The VMM's event loop receives that signal and shuts down the VM process gracefully. From the guest's perspective, the CPU stops; from the host's perspective, the `firecracker` binary exits cleanly. The control register keeps two bits set permanently: `CB_POST_OK = 0x04` (Power-On Self Test passed) and `CB_KBD_INT = 0x01` (keyboard interrupt enabled), which is sufficient to prevent the Linux keyboard driver from looping waiting for POST completion.

The only scan codes the emulator produces are for Ctrl+Alt+Del: `KEY_CTRL = 0x0014`, `KEY_ALT = 0x0011`, `KEY_DEL = 0xE071`. The emulator is not a general PS/2 keyboard; it cannot type. From v1.8.0 onward, it appears in the ACPI DSDT as `_SB_.PS2_` with HID `PNP0303`, with I/O resources at `0x0060` (size 1) and `0x0064` (size 1, the latter described in the source as "Fake a command port so Linux stops complaining").

PICs, PIT, and LAPIC: KVM's In-Kernel Devices

Three more legacy devices are present in every Firecracker VM, but Firecracker does not emulate them itself — KVM does, in the kernel, before any userspace instruction executes. Firecracker's own design document describes them this way: "In addition to the Firecracker provided device models, guests also see the Programmable Interrupt Controllers (PICs), the I/O Advanced Programmable Interrupt Controller (IOAPIC), and the Programmable Interval Timer (PIT) that KVM supports."

Creating the interrupt fabric. A single ioctl, `KVM_CREATE_IRQCHIP` (`_IO(0xAE, 0x60)`), instantiates two cascaded i8259A PICs — master at ports `0x20` / `0x21`, slave at ports `0xA0` / `0xA1` — and an IOAPIC. It also arranges for every subsequently-created vCPU to have a Local APIC. After this call, GSIs 0–15 are

routed to both the PIC and the IOAPIC; GSIs 16–23 go to the IOAPIC only. Critically, `KVM_CREATE_IRQCHIP` must be called *before* `KVM_CREATE_VCPU` on x86_64; the ordering is enforced by KVM and documented in the KVM API under §4.24.

Firecracker checks for the required KVM capabilities — `KVM_CAP_PIT2` and `KVM_CAP_PIT_STATE2` — during startup in `src/vmm/src/arch/x86_64/kvm.rs`. Missing either capability aborts VM creation. Then `setup_irqchip()` in `src/vmm/src/arch/x86_64/vm.rs` calls `create_irq_chip()` followed immediately by `create_pit2(kvm_pit_config { flags: KVM_PIT_SPEAKER_DUMMY, ..Default::default() })`.

The PIT. `KVM_CREATE_PIT2` (`_IOW(0xAE, 0x77, struct kvm_pit_config)`) creates the i8254 Programmable Interval Timer. Counter 0 at port `0x40` drives system timer IRQ 0; counter 1 at `0x41` is vestigial (originally DRAM refresh); counter 2 at `0x42` gates the PC speaker via port `0x61`. The tick rate is `PIT_TICK_RATE = 1193182 Hz` (defined in `include/linux/timex.h`). The `KVM_PIT_SPEAKER_DUMMY` flag in the `flags` field tells KVM to emulate the speaker port at `0x61` in-kernel, avoiding a userspace VM exit every time Linux probes it.

Why the PIT matters for TSC calibration. The PIT is not just a timer; it is the ruler against which the kernel measures the TSC's frequency. `arch/x86/kernel/tsc.c` contains two calibration functions, `pit_calibrate_tsc()` and `quick_pit_calibrate()`. Both gate counter 2 via port `0x61`, program the measurement latch via port `0x43`, then poll the PIT MSB in a tight loop (up to `CAL_PIT_LOOPS = 1000` or `CAL2_PIT_LOOPS = 5000` iterations) measuring elapsed TSC ticks against a 10 ms window (`CAL_MS = 10`). The formula is: `kHz = ((t2 − t1) ∗ PIT_TICK_RATE) / (latch ∗ 1000)` where `CAL_LATCH = PIT_TICK_RATE / (1000 / CAL_MS)`.

Both functions guard on `has_legacy_pic()`. `quick_pit_calibrate()` returns 0 immediately when no legacy PIC is present. `pit_calibrate_tsc()` instead falls back to a `udelay`-based wait but still skips the PIT measurement loop. Either way, removing the PIT pushes the kernel onto CPUID-based or udelay-based frequency estimation — slower and less precise. Preserving the PIT preserves the fast, accurate TSC calibration path.

Snapshot and restore. Because the interrupt controllers are in-kernel, saving their state requires ioctls. Firecracker's `save_state()` calls `get_irqchip()` with `KVM_GET_IRQCHIP` three times — once each for `KVM_IRQCHIP_PIC_MASTER`, `KVM_IRQCHIP_PIC_SLAVE`, and `KVM_IRQCHIP_IOAPIC` — writing into a `struct kvm_irqchip { chip_id; chip }`. The restore path calls `set_irqchip()` with `KVM_SET_IRQCHIP` three times in the same order. This symmetric protocol is one reason Firecracker's snapshotting is fast: no in-process state machine needs serializing, only three kernel structures.

Fixed MMIO addresses. The IOAPIC sits at `IOAPIC_ADDR = 0xFEC0_0000` and the LAPIC at `APIC_ADDR = 0xFEE0_0000` (from `src/vmm/src/arch/x86_64/layout.rs`). KVM also needs a protected TSS at `0xFFFB_D000` (the KVM TSS region), which is placed outside any guest-accessible memslot.

GSI allocation. With the full interrupt fabric in place, Firecracker allocates GSIs as follows (from `layout.rs`):

Range	Owner
GSI 0–4	Reserved; COM1 = GSI 4, i8042 keyboard = GSI 1, timer IRQ 0 = GSI 0
GSI 5–23	virtio-mmio device slots
GSI 24–4095	MSI (PCIe, since v1.13.0)

The MMIO Bus and Device Routing

Having enumerated the legacy devices, the next question is mechanical: when the guest executes `IN 0x3F8, AL` or writes to a virtio queue-notify register, how does the instruction reach its emulator?

Exits to Userspace

Under VMX, two classes of instruction produce exits that land in the VMM. An `IN` or `OUT` instruction generates exit reason 30 (`EXIT_REASON_IO_INSTRUCTION`). A guest memory access to an address with no backing EPT mapping generates an EPT violation (exit reason 48) or EPT misconfiguration (exit reason 49); if no in-kernel handler claims the fault, KVM promotes it to a `KVM_EXIT_MMIO` and returns to userspace.

Both classes surface through `KVM_RUN` (`_IO(0xAE, 0x80)`). After `KVM_RUN` returns, the VMM reads `kvm_run.exit_reason`. For I/O port accesses it is `KVM_EXIT_IO = 2`; for MMIO accesses it is `KVM_EXIT_MMIO = 6`. The sub-structs in the `kvm_run` page describe what happened:

```
/* KVM_EXIT_IO */
struct {
    __u8  direction;   /* KVM_EXIT_IO_IN=0, KVM_EXIT_IO_OUT=1 */
    __u8  size;        /* operand size: 1, 2, or 4 bytes */
    __u16 port;        /* I/O port number */
    __u32 count;       /* repetition count for INS/OUTS */
    __u64 data_offset; /* byte offset into kvm_run mapping to the data buffer */
} io;

/* KVM_EXIT_MMIO */
struct {
    __u64 phys_addr;  /* guest physical address */
    __u8  data[8];    /* up to 8 bytes of data */
    __u32 len;        /* access width in bytes */
    __u8  is_write;   /* 1 = write, 0 = read */
} mmio;
```

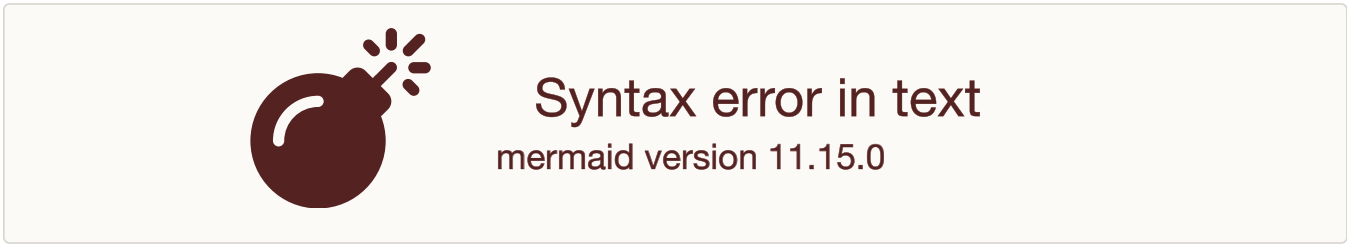
Source: `include/uapi/linux/kvm.h` in torvalds/linux.

MMIO space is not configured explicitly. Any guest physical address (GPA) range not covered by a `KVM_SET_USER_MEMORY_REGION` memslot (`_IOW(0xAE, 0x46, struct kvm_userspace_memory_region)`) produces `KVM_EXIT_MMIO` on access. Firecracker calls `KVM_SET_USER_MEMORY_REGION` to map RAM (and, from v1.8.0, the page holding the RSDP); everything else is MMIO by omission.

Firecracker's Software Bus

The exit reason alone does not route the access. The VMM needs a data structure that maps port or address ranges to device emulators. Firecracker implements its own `Bus` in `src/vmm/src/vstate/bus.rs`. It is a `RwLock<BTreeMap<BusRange, Weak<dyn BusDeviceSync>>>`: a sorted tree mapping half-open address ranges to device references. Lookup uses the B-tree's predecessor operation — `range(..= BusRange::new(addr, 1)).next_back()` — and then checks that the address falls within the range's end; the whole lookup is O(log n) in the number of registered devices.

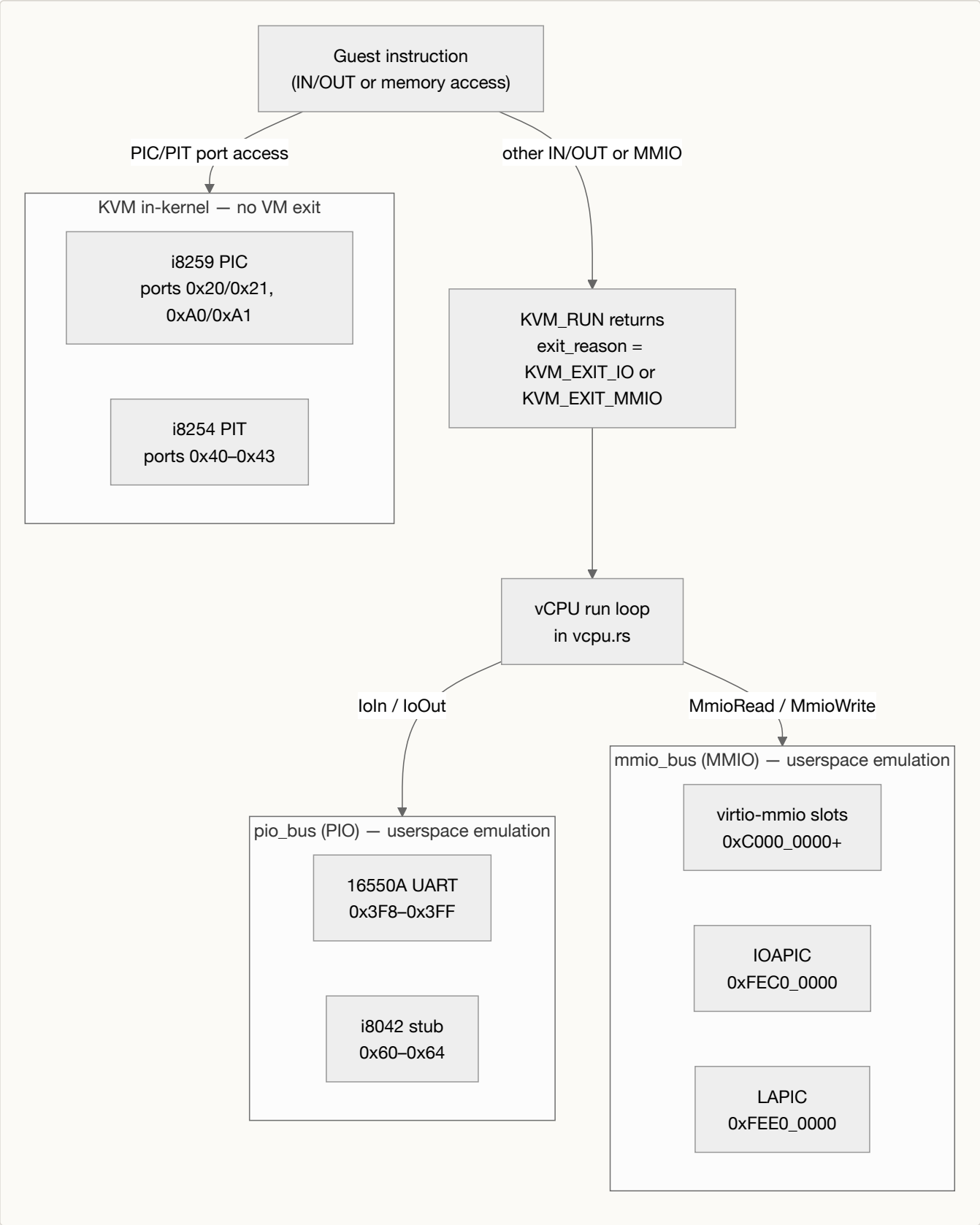
Each vCPU holds two buses: `pio_bus: Option<Arc<Bus>>` for I/O port accesses (x86_64 only) and `mmio_bus: Option<Arc<Bus>>` for MMIO accesses. The vCPU run loop dispatches exits like this:



(Source: `src/vmm/src/vstate/vcpu.rs`.)

Devices are stored as `Weak<dyn BusDeviceSync>` — the bus does not keep devices alive; the VMM's owner struct holds the `Arc`. The `BusDevice` trait exposes `read(&mut self, base: u64, offset: u64, data: &mut [u8])` and `write(&mut self, base: u64, offset: u64, data: &[u8]) -> Option<Arc<Barrier>>`. An unresolved address is logged as a `warn!` but returns `VcpuEmulation::Handled`, so an out-of-range access does not crash the VM.

The relationship between the exit path, the bus, and the device emulators looks like this:


Guest instruction
(IN/OUT or memory access)
PIC/PIT port access
other IN/OUT or MMIO
KVM in-kernel — no VM exit
i8259 PIC
ports 0x20/0x21,
0xA0/0xA1
i8254 PIT
ports 0x40–0x43
KVM_RUN returns
exit_reason =
KVM_EXIT_IO or
KVM_EXIT_MMIO
vCPU run loop
in vcpu.rs
IoIn / IoOut
MmioRead / MmioWrite
pio_bus (PIO) — userspace emulation
16550A UART
0x3F8–0x3FF
i8042 stub
0x60–0x64
mmio_bus (MMIO) — userspace emulation
virtio-mmio slots
0xC000_0000+
IOAPIC
0xFEC0_0000
LAPIC
0xFEE0_0000

The Fast Path: KVM_IOEVENTFD

Virtio queue kick notifications would generate a `KVM_EXIT_MMIO` on every `VIRTIO_MMIO_QUEUE_NOTIFY` write (register offset `0x050` from the device's base address). At high I/O rates, round-tripping through `KVM_RUN` for each kick is expensive. `KVM_IOEVENTFD` (`_IOW(0xAE, 0x79, struct kvm_ioeventfd)`) bypasses this. It registers an `eventfd` with KVM for a specific address range; when the guest writes to that range, KVM signals the eventfd in-kernel and immediately re-enters the guest. The device backend thread wakes from a `read(eventfd_fd)` and processes the queue without the VMM loop ever seeing the exit. This is how Firecracker wires virtio queue kicks: driver writes `VIRTIO_MMIO_QUEUE_NOTIFY` → KVM signals eventfd → backend thread drains the queue → no userspace round-trip.

The Physical Address Map

With RAM, MMIO, and the fixed IOAPIC and LAPIC addresses laid out, the guest physical address space for a typical Firecracker VM looks like this:

```
flowchart TB
  subgraph gpa["Guest Physical Address Space (x86_64)"]
    lo["0x0000_0000 — RAM<br/>(up to ~3 GiB)"]
    mmio32["0xC000_0000 — 32-bit MMIO window (1 GiB)<br/>virtio-mmio slots, 4 KiB each"]
    ioapic_block["0xFEC0_0000 — IOAPIC"]
    lapic_block["0xFEE0_0000 — LAPIC"]
    hi_ram["0x1_0000_0000 — High RAM<br/>(if guest > 3 GiB)"]
    mmio64["256 GiB — 64-bit MMIO window (256 GiB)"]
  end
```

Each virtio-mmio slot is 4 KiB (`MMIO_LEN = 0x1000`), starting at `BOOT_DEVICE_MEM_START = 0xC000_0000` for the first (boot) device and `MEM_32BIT_DEVICES_START = 0xC000_1000` for subsequent ones. There is no PCI configuration space, no PCIe ECAM window, and no Option ROM area — just RAM, virtio slots, and the two APIC regions.

What Firecracker Drops

No BIOS

Classical x86 boot requires the CPU to start in 16-bit real mode, read a boot sector from disk, hand control to a bootloader, and eventually transition to 64-bit protected mode — all mediated by a BIOS ROM that QEMU or other VMMs typically provide as `bios.bin` or `OVMF.fd`. Firecracker skips the entire stack.

Instead, Firecracker constructs a `struct boot_params` (the "zero page" of the Linux x86 boot protocol, currently version 2.15 as of kernel 5.5) and places it at guest physical address `0x7000`. It sets the `%rsi` register to point to that address and jumps directly to the 64-bit kernel entry point at the load address

plus `0x200` — the standard offset for a bzImage's 64-bit entry stub. No real-mode code executes, no BIOS ROM is mapped, and the `boot_params` fields for legacy I/O devices are not used; the serial console, for instance, is configured entirely via kernel command line, not via `setup_header`.

Alternatively, Firecracker can use the Xen PVH Direct Boot ABI (added in Firecracker v1.12.0). In the PVH path, Firecracker writes an `hvm_start_info` structure at `PVH_INFO_START = 0x6000` with magic value `XEN_HVM_START_MAGIC_VALUE = 0x336e_c578` in `%rbx`. The kernel must be compiled with `CONFIG_PVH=y`, available since Linux 5.0; the ELF binary then contains a PVH entry point in a `PT_NOTE` segment. Both paths eliminate firmware entirely; they differ only in the handshake structure the kernel expects to find before its first instruction.

No PCI Bus

Firecracker's virtio devices use the MMIO transport defined in virtio specification §4.2 (`OASIS virtio 1.2, Committee Specification 01`), not the PCI transport. There is no PCI host bridge, no PCI configuration space mechanism (neither CF8/CFC port-IO nor PCIe ECAM MMIO), and no PCI enumeration.

The MMIO transport has no self-describing enumeration: a device at a given address does not announce its type or existence on the bus. Discovery happens through side channels. Before Firecracker v1.8.0, this meant kernel command- line slugs of the form `virtio_mmio.device=512@0xC0001000:6`, one per device, injected into the kernel command line by the VMM at boot. From v1.8.0 onward, an ACPI DSDT table enumerates virtio devices with their MMIO addresses and assigned GSIs, so the guest kernel does not need the command-line hint.

PCI was added as an opt-in in Firecracker v1.13.0 via `--enable-pci`. When enabled, VirtIO devices use a PCI VirtIO transport instead; MMIO remains the default. Skipping the PCI host bridge and configuration space mechanism removes a substantial slice of emulated attack surface and eliminates the enumeration overhead that PCI scanning adds to early boot.

No ACPI Power Management

Firecracker added basic ACPI table support in v1.8.0: an FADT, XSDT, MADT (for the LAPIC and IOAPIC), and DSDT describing virtio and legacy devices. The RSDP pointer sits at `RSDP_ADDR = 0x000E_0000`. But the FAQ states the boundary explicitly: "Firecracker does not virtualize power management (e.g. there is no ACPI PM support)."

ACPI S3 (suspend to RAM), S4 (hibernate), and S5 (soft-off) are not available. Reboot is handled by the i8042 `CMD_RESET_CPU = 0xFE` path described above. Shutdown initiated from inside the guest — for instance, `poweroff` — does not trigger a clean power-off sequence. The Firecracker process continues running until an external caller sends the `SendCtrlAltDel` API event, which injects a Ctrl+Alt+Del scan code sequence into the guest, ultimately causing the kernel to reboot via the i8042 path. Before v1.8.0, device enumeration used an MPTable; from v1.8.0 that path is deprecated, with removal planned for v2.0.

The attack surface that remains is deliberately auditable: the UART, i8042, and PIT emulators each fit in a few hundred lines of Rust. Chapter 13 covers the `jailer` process, which further constrains what the VMM can reach even if one of those emulators is compromised.

QEMU microvm: The Same Destination, Different Road

QEMU's `microvm` machine type (`-machine microvm`) was introduced in QEMU 4.2 in late 2019. The QEMU documentation describes it as "a machine type inspired by Firecracker and constructed after its machine model," and the structural similarity is clear: a single ISA bus, no PCI by default, no ACPI in the original release, legacy devices kept only where necessary. The differences are mostly of degree, and understanding them sharpens what is genuinely necessary in any minimal machine model versus what is a Firecracker-specific choice.

The Bus Fabric

QEMU microvm's only bus is a single ISA bus. Onto that bus, a set of legacy devices can be optionally attached: the i8259 PIC pair, the i8254 PIT, an MC146818 RTC, and one ISA serial port. The LAPIC and IOAPIC are always present when KVM is in use; `kernel-irqchip=split` is the default KVM irqchip mode for microvm, meaning the LAPIC lives in KVM's kernel module while the IOAPIC is handled by QEMU userspace.

virtio-mmio Slots

QEMU microvm provides 8 virtio-mmio transport slots by default (`mms->virtio_num_transports = 8` in `hw/i386/microvm.c`). Each slot is 512 bytes wide (smaller than Firecracker's 4 KiB per slot) at a base address of `VIRTIO_MMIO_BASE = 0xfeb00000` . Slot `i` sits at `0xfeb00000 + i * 512` . The default IRQ base is `mms->virtio_irq_base = 5` , so slots 0–7 use GSIs 5–12.

With a secondary IOAPIC (`ioapic2`), the virtio IRQ base moves to `IO_APIC_SECONDARY_IRQBASE` (24), the slot count grows to `IOAPIC_NUM_PINS` (24), and PCIe (when enabled) takes IRQs 12–15. Other fixed MMIO addresses from `include/hw/i386/microvm.h` : the ACPI Generic Event Device (GED) at `0xfea00000` on IRQ 9, optional xHCI USB at `0xfe900000` on IRQ 10, and the PCIe ECAM window at `0xe0000000` (size 256 MiB) with a MMIO window at `0xc0000000` (size 512 MiB) when PCIe is on.

Firmware

This is where QEMU microvm and Firecracker diverge most sharply. QEMU microvm supports direct kernel loading via `-kernel` — the QEMU documentation describes it as a machine type that "needs to be run using a host-side kernel and, optionally, an initrd image." But the firmware stub still executes. In `hw/i386/microvm.c` , `x86_bios_rom_init()` is called unconditionally as long as IGVM mode is not active: with ACPI disabled it maps `qboot.rom` , with ACPI enabled it maps `bios-microvm.bin` . The `-kernel` flag tells QEMU where to load the kernel image, but it does not suppress the ROM. The guest CPU starts in the firmware stub, which then hands off to the kernel.

Firecracker takes the opposite approach: the VMM constructs `boot_params` directly, sets `%rsi` to point to it, and jumps to the 64-bit kernel entry point. No ROM is mapped, no firmware code executes, and the guest CPU's first instruction is the kernel's own. qboot is purpose-built for speed — it typically adds only a few tens of milliseconds — but it still represents firmware-controlled code running in the guest before the kernel. Firecracker eliminates that phase entirely.

ACPI

QEMU 4.2 shipped microvm without ACPI. QEMU 5.2 added it. The tables are compact: APIC at 78 bytes, DSDT at 482 bytes, FACP at 268 bytes — under 1 KiB total, growing to roughly 3,130 bytes for the DSDT when PCIe is enabled (per Gerd Hoffmann's 2020 blog post on `kraxel.org`). The DSDT declares each active virtio-mmio slot with its MMIO address and GSI, so no command-line slugs are needed.

When ACPI is disabled, QEMU handles device discovery by patching the guest kernel command line automatically: `microvm_get_mmio_cmdline()` in `hw/i386/microvm.c` appends `virtio_mmio.device=512@0x<addr>:<irq>` for each active slot. This behavior is controlled by the machine option `auto-kernel-cmdline` (on by default).

Shutdown

Without ACPI PM and without a PS/2 keyboard (both of which are optional in microvm), there is no standard shutdown path. QEMU microvm's recommended approach is a CPU triple-fault, which QEMU treats as a reboot or shutdown trigger. The kernel parameter `reboot=t` prioritizes the triple-fault path. This is the mirror image of Firecracker's `reboot=k` strategy: both avoid ACPI PM, but Firecracker routes through the i8042 while QEMU microvm, when the i8042 is absent, routes through a deliberate fault.

Side by Side

Property	QEMU microvm	Firecracker
PCI bus	None (QEMU docs describe microvm as having no PCI/PCIe)	None by default; optional PCIe (v1.13.0+)
ACPI	Added in QEMU 5.2; includes PM framework	Added in v1.8.0; no PM
Firmware	`qboot.rom` (no ACPI) or `bios-microvm.bin` (with ACPI)	None; kernel loaded directly
virtio transport	virtio-mmio; 8 slots at `0xfeb00000`, 512 B each	virtio-mmio; 4 KiB slots from `0xC000_0000`
ISA serial	Optional; always firmware-visible	Present; disabled in production builds
Shutdown	CPU triple-fault (`reboot=t`)	i8042 `CMD_RESET_CPU=0xFE` (`reboot=k`)
Device enumeration	Command-line injection or ACPI DSDT	Command-line injection, MPTable (deprecated), or ACPI DSDT (v1.8.0+)
Introduced	QEMU 4.2, late 2019	Open-sourced November 2018 (v0.11.0)

The structural gap is the firmware stub — a few tens of milliseconds of vendor-controlled code that QEMU microvm runs before every kernel, and that Firecracker never maps at all. The next chapter examines what the jailer does with the attack surface that remains after the machine model has been stripped this far down.

Sources And Further Reading

- Firecracker legacy device manager (constants, ACPI AML for COM1 and PS/2): `src/vmm/src/device_manager/legacy.rs` https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/device_manager/legacy.rs
- Firecracker i8042 emulator (command constants, status/control bits, scan codes): `src/vmm/src/devices/legacy/i8042.rs` https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/devices/legacy/i8042.rs
- vm-superio 16550A emulation (`FIFO_SIZE`, `IIR_FIFO_BITS`, register map): `vm-superio/src/serial.rs` https://github.com/rust-vmm/vm-superio/blob/main/vm-superio/src/serial.rs

- Firecracker x86_64 memory layout (MMIO base, slot size, GSI ranges, IOAPIC/LAPIC addresses): `src/vmm/src/arch/x86_64/layout.rs` https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/arch/x86_64/layout.rs
- Firecracker irqchip and PIT setup (`setup_irqchip`, `KVM_PIT_SPEAKER_DUMMY`): `src/vmm/src/arch/x86_64/vm.rs` https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/arch/x86_64/vm.rs
- Firecracker required KVM capabilities check: `src/vmm/src/arch/x86_64/kvm.rs` https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/arch/x86_64/kvm.rs
- Firecracker `Bus` struct and `BusDevice` trait: `src/vmm/src/vstate/bus.rs` https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/vstate/bus.rs
- Firecracker vCPU run loop and exit dispatch: `src/vmm/src/vstate/vcpu.rs` https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/vstate/vcpu.rs
- Firecracker design document (device list, thread model, production serial disable): https://github.com/firecracker-microvm/firecracker/blob/main/docs/design.md
- Firecracker FAQ (`reboot=k`, "6 emulated devices," no ACPI PM, boot-time guarantee): https://github.com/firecracker-microvm/firecracker/blob/main/FAQ.md
- Firecracker v1.8.0 release notes (ACPI tables added; RSDP placement; MPTable deprecated): https://github.com/firecracker-microvm/firecracker/releases/tag/v1.8.0
- Firecracker v1.12.0 release notes (PVH boot mode added; `CONFIG_PVH=y`, Linux 5.0+): https://github.com/firecracker-microvm/firecracker/releases/tag/v1.12.0
- Firecracker v1.13.0 release notes (optional PCI transport via `--enable-pci`): https://github.com/firecracker-microvm/firecracker/releases/tag/v1.13.0
- KVM UAPI header (`KVM_EXIT_IO`, `KVM_EXIT_MMIO`, `kvm_run` sub-structs, `KVM_CREATE_IRQCHIP`, `KVM_CREATE_PIT2`, `KVM_IOEVENTFD`): https://github.com/torvalds/linux/blob/master/include/uapi/linux/kvm.h
- KVM API documentation (§4.24 `KVM_CREATE_IRQCHIP`, §4.71 `KVM_CREATE_PIT2`, §4.59 `KVM_IOEVENTFD`, §4.35 `KVM_SET_USER_MEMORY_REGION`): https://docs.kernel.org/virt/kvm/api.html
- Linux `timex.h` (`PIT_TICK_RATE = 1193182 Hz`): https://github.com/torvalds/linux/blob/master/include/linux/timex.h
- Linux `tsc.c` (`pit_calibrate_tsc`, `quick_pit_calibrate`, `CAL_PIT_LOOPS`, `has_legacy_pic`): https://github.com/torvalds/linux/blob/master/arch/x86/kernel/tsc.c
- Linux serial console documentation (`earlycon`, `uart8250,io,0x3f8`): https://docs.kernel.org/admin-guide/serial-console.html

- Linux x86 boot protocol v2.15: https://www.kernel.org/doc/html/v6.1/x86/boot.html
- OASIS virtio 1.2 specification (§4.2 MMIO transport, §4.2.1 discovery, §4.2.2 register map): https://docs.oasis-open.org/virtio/virtio/v1.2/cs01/virtio-v1.2-cs01.html
- Linux virtio-mmio UAPI header (all register offsets including `VIRTIO_MMIO_QUEUE_NOTIFY=0x050`): https://raw.githubusercontent.com/torvalds/linux/master/include/uapi/linux/virtio_mmio.h
- QEMU microvm documentation: https://www.qemu.org/docs/master/system/i386/microvm.html
- QEMU `hw/i386/microvm.c` (slot count, base address, IRQ base, firmware selection, `microvm_get_mmio_cmdline`): https://github.com/qemu/qemu/blob/master/hw/i386/microvm.c
- QEMU `include/hw/i386/microvm.h` (fixed MMIO addresses: GED, xHCI, PCIe window): https://github.com/qemu/qemu/blob/master/include/hw/i386/microvm.h
- Gerd Hoffmann (QEMU maintainer), "QEMU microvm and ACPI" (ACPI table sizes, firmware selection, `bios-microvm.bin`): https://www.kraxel.org/blog/2020/10/qemu-microvm-acpi/
- Stefano Garzarella (QEMU developer), boot time measurement methodology for microvm and qboot (phase-by-phase tracing, virtme tooling): https://stefano-garzarella.github.io/posts/2019-08-24-qemu-linux-boot-time/
- rust-vmm `vm-device` bus abstractions (`IoManager`, `DevicePio`, `DeviceMmio`): https://github.com/rust-vmm/vm-device/blob/main/src/bus/mod.rs https://github.com/rust-vmm/vm-device/blob/main/src/device_manager.rs

PART IV — FIRECRACKER END TO END

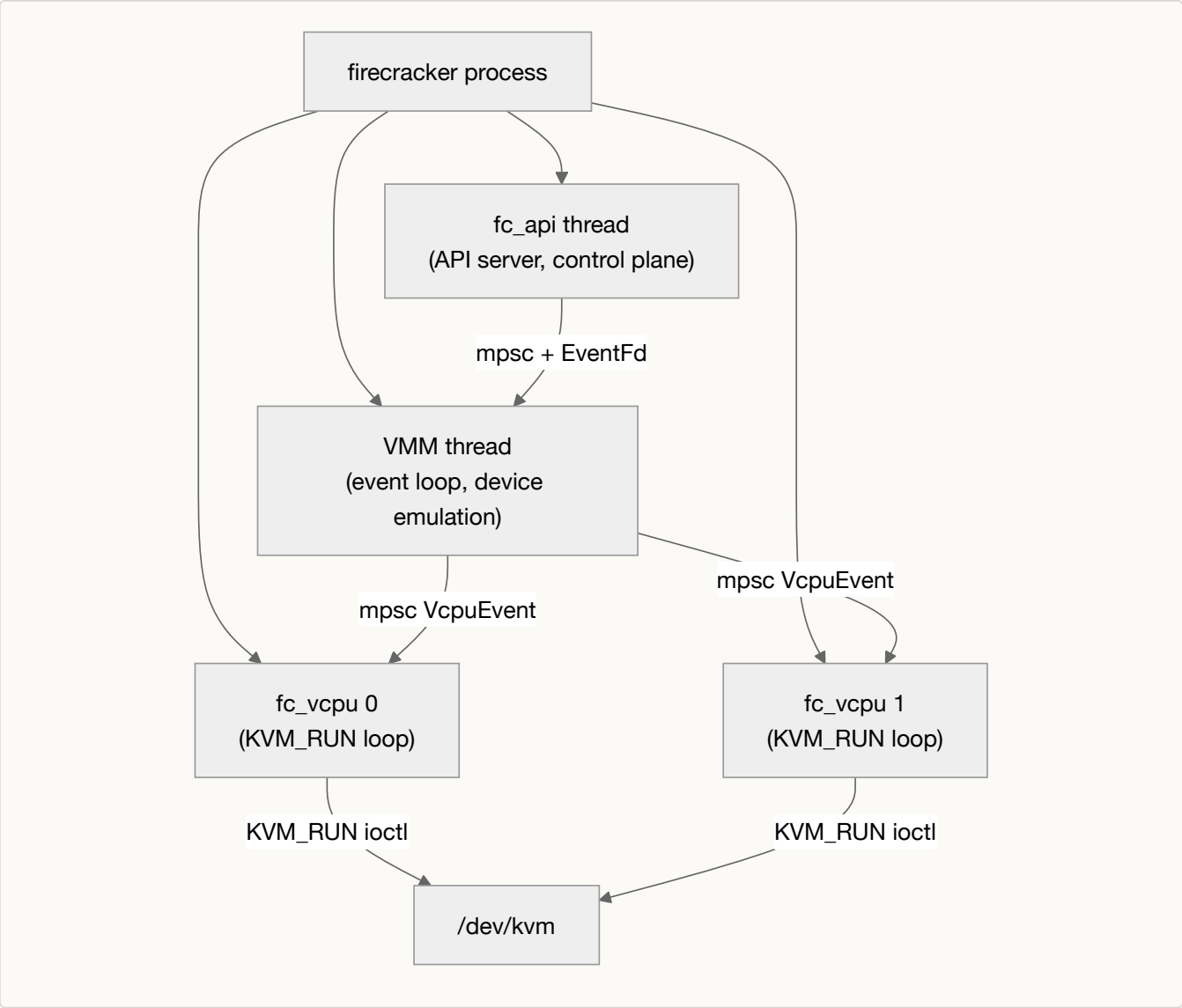
Chapter 13: Firecracker Architecture

What does one `firecracker` process contain, and how are its parts connected? The process must serve a REST API, run one or more guest vCPU loops, emulate a minimal device set, and do all of it while keeping the guest's influence over the host kernel as narrow as a handful of ioctls. This chapter traces the architecture that satisfies those constraints.

One Process, One MicroVM

Firecracker's design document states the invariant plainly: "Each Firecracker process encapsulates one and only one microVM." There is no multiplexing, no VM table, no daemon that routes requests to a pool. Running N microVMs means N `firecracker` processes. This is a deliberate constraint, not an implementation shortcut. A single process failure cannot cascade to a neighboring VM; the host kernel's process isolation is part of the security boundary, complementing the hardware VMX enforcement and the jailer's chroot.

Inside that one process, three categories of OS thread do distinct work.



The API thread is named `fc_api` via `thread::Builder::new().name("fc_api")`. It runs the HTTP server over a Unix domain socket and handles every configuration request, but it never touches guest execution directly. The VMM thread is the hub: it owns the KVM `VmFd`, all device backends, the MMDS metadata service, and the `EventManager` epoll loop. The vCPU threads — one per guest CPU core, named `fc_vcpu 0`, `fc_vcpu 1`, and so on using `format!("fc_vcpu {}", self.kvm_vcpu.index)` — do nothing but call `KVM_RUN` and service the exits that return from it. The vCPU threads are the part that runs untrusted code, and Firecracker's security design treats them accordingly: they receive the most restrictive seccomp-BPF filter of any thread in the process.

The REST API Over a Unix Socket

Every Firecracker configuration operation arrives through a single Unix domain socket. The socket is never TCP, always a SOCK_STREAM Unix socket. A common convention is `/run/firecracker.socket`; the `--api-sock` command-line flag sets the path explicitly. When the jailer is in use, the socket appears inside the chroot at `<chroot_base>/<exec_file_name>/<id>/root/<api-sock>`, which from the host looks like `/srv/jailer/firecracker/<id>/root/run/firecracker.socket`.

Firecracker does not use a standard Rust HTTP crate. It uses `micro_http`, an in-house crate maintained by the `firecracker-microvm` GitHub organization at `https://github.com/firecracker-microvm/micro-http`. It is referenced in `Cargo.toml` as a git dependency, not a crates.io package. `micro_http` implements HTTP/1.0 and HTTP/1.1 with no chunked transfer encoding and no compression; it enforces a configurable maximum request body size via `server.set_payload_max_size(limit)`, returning HTTP 400 on violation. The API socket is available within at most 8 CPU-milliseconds of process start — wall-clock typically around 12 ms, with observed range 6–60 ms — because the API thread and socket setup complete before the VMM thread touches KVM.

The API conforms to an OpenAPI specification kept in-tree at `src/firecracker/swagger/firecracker.yaml`. `SPECIFICATION.md` says: "The API socket is always available and the API conforms to the in-tree Open API specification." All resources are stable across patch versions; the specification is the normative contract.

How the API Thread Talks to the VMM Thread

The two threads share two `std::sync::mpsc` channels and one `EventFd`:

```
API thread                                VMM thread
   |                                          |
   |--- Sender<ApiRequest> -------------->  |   (boxed VmmAction)
   |--- EventFd (api_event_fd).write() --> |   (wake-up signal)
   |<-- Sender<ApiResponse> -------------- |   (reply)
```

`ApiServer::new()` takes three arguments: an `mpsc::Sender<ApiRequest>` to forward requests into, an `mpsc::Receiver<ApiResponse>` to read replies from, and the `EventFd` that wakes the VMM's event loop. When a request arrives on the socket, the API thread sends the boxed action down the channel and writes to the `EventFd`. The VMM thread is blocking in its `EventManager` epoll loop; the `EventFd` write wakes it, and it drains the request receiver synchronously. The reply comes back on the reverse channel. From the API caller's perspective this looks like a synchronous RPC; under the hood it is two decoupled threads producing request-response semantics through a channel pair.

A slow API request — configuring 32 block devices before boot, for example — does not pause a running vCPU. The two threads are independent.

The VMM Thread: Events and Devices

The VMM thread owns the in-memory `Vmm` struct and the `KvmVm` it contains. Its central primitive is `EventManager`, an epoll-based event loop. Every device backend and the VMM itself register as event subscribers. When a vCPU exits with `KVM_EXIT_MMIO` — the exit reason is 6 in `<linux/kvm.h>` — the exit delivers the MMIO address and data through the `kvm_run` struct; the vCPU thread that took the exit dispatches directly to the device handler via `mmio_bus.read()` or `mmio_bus.write()` on the `Peripherals` struct attached to that vCPU, without waking the VMM thread.

The devices the VMM thread owns are the five virtio backends — Net, Block, Vsock, Balloon, and Entropy — plus the Microvm Metadata Service and the legacy device model. The legacy model is minimal: an i8042 PS/2 controller (used on x86-64 exclusively for CPU reset signaling) and a serial UART, both implemented via the `vm-superio` crate. The UART is the 16550A-compatible `Serial` struct; the i8042 is the `I8042Device` struct. Neither of these is in the hot path once the guest has booted; they exist to provide the reset mechanism and, when enabled, a serial console for debugging.

When a vCPU thread finishes its work — either because guest execution completed or because it received a shutdown event — it writes to an `exit_evt: EventFd` that was passed at construction time as `Vcpu::new(..., exit_evt)`. The VMM's `EventManager` subscribes to this fd via `vcpus_exit_evt()`. That is how the VMM detects an unexpected vCPU exit without polling.

vCPU Threads: The KVM_RUN Loop

`KvmVm::start_vcpus()` calls `vcpu.start_threaded(...)` for each vCPU. Inside `start_threaded`, the spawn looks like:

```
thread::Builder::new()
    .name(format!("fc_vcpu {}", self.kvm_vcpu.index))
    .spawn(move || { ... })
```

vCPU threads start in a paused state. `Vmm::resume_vm()` must be called to release them into guest execution. That call is the hard boundary between the pre-boot phase and the running phase; before it, configuration is still mutable.

Inside the vCPU thread's loop, the call is `self.kvm_vcpu.fd.run()`, which issues the `KVM_RUN` vcpu ioctl — `_IO(KVMIO, 0x80)` — on the vcpu fd. The ioctl does not return until the guest causes a VM exit. The exit reason lives in the `kvm_run` struct that KVM maps into the VMM's address space via `mmap` on the vcpu fd; the mapping size comes from `KVM_GET_VCPU_MMAP_SIZE` (`_IO(KVMIO, 0x04)`). Firecracker reads this through `kvm-ioctls`'s `KvmRunWrapper`, which exposes the struct fields safely from Rust.

Firecracker's `handle_kvm_exit()` dispatches on a `VcpuExit` enum (from `kvm-ioctls`) that covers the following exits:

Exit	`kvm.h` constant	Action
`MmioRead(addr, data)`	`KVM_EXIT_MMIO` (6)	Read device register, fill data buffer
`MmioWrite(addr, data)`	`KVM_EXIT_MMIO` (6)	Write device register
`SystemEvent(RESET)`	`KVM_EXIT_SYSTEM_EVENT` (24)	Return `VcpuEmulation::Stopped`
`SystemEvent(SHUTDOWN)`	`KVM_EXIT_SYSTEM_EVENT` (24)	Return `VcpuEmulation::Stopped`
`FailEntry`	`KVM_EXIT_FAIL_ENTRY` (9)	Log and stop
`InternalError`	(internal)	Log and stop
`Hlt`	`KVM_EXIT_HLT` (5)	`UnhandledKvmExit` error
`Shutdown`	`KVM_EXIT_SHUTDOWN` (8)	`UnhandledKvmExit` error

`KVM_EXIT_HLT` and `KVM_EXIT_SHUTDOWN` are treated as errors rather than graceful shutdowns because Firecracker expects the guest to use `KVM_EXIT_SYSTEM_EVENT` for clean power-off. A `HLT` that reaches the VMM means no device has consumed the halt — something unexpected happened.

To interrupt a running vCPU from another thread, Firecracker calls `self.kvm_vcpu.fd.set_kvm_immediate_exit(1)`. This sets the `immediate_exit` field in the shared `kvm_run` struct to 1, which causes the next (or current) `KVM_RUN` to return with `EINTR`. The vCPU thread then drains its command channel — an `mpsc::Receiver<VcpuEvent>` — and acts on whatever the VMM thread sent (Pause, Resume, SaveState, and so on).

The KVM Ioctl Hierarchy

The three fd types and their key ioctls form a strict hierarchy. You cannot issue a vCPU ioctl without first having a VM fd; you cannot have a VM fd without first having opened `/dev/kvm`.

> ***Note:*** *The following sequence opens `/dev/kvm` and creates kernel-managed resources. Do not run it on a shared machine; it requires read-write access to `/dev/kvm` and the `CAP_SYS_ADMIN` capability (or membership in the `kvm` group on most distributions).*

Level	fd	Key ioctl	Encoding	Purpose
System	`/dev/kvm`	`KVM_CREATE_VM`	`_IO(KVMIO, 0x01)`	Returns a VM fd
VM	VM fd	`KVM_SET_USER_MEMORY_REGION`	`_IOW(KVMIO, 0x46, ...)`	Maps host memory as guest RAM
VM	VM fd	`KVM_CREATE_VCPU`	`_IO(KVMIO, 0x41)`	Returns a vcpu fd per core
vCPU	vcpu fd	`KVM_RUN`	`_IO(KVMIO, 0x80)`	Enters guest; returns on each exit

`KVM_GET_API_VERSION` returns 12; this value has been stable since KVM's introduction and Firecracker checks it at startup.

`KVM_SET_USER_MEMORY_REGION` takes a `kvm_userspace_memory_region` struct:

Slots may not overlap in guest physical address space. Setting `KVM_MEM_LOG_DIRTY_PAGES` in `flags` enables the dirty-page bitmap that diff snapshots depend on — a point the state machine section returns to.

The rust-vmm Crates

Firecracker is not a monolith built from scratch. It is assembled from a set of shared crates maintained under the `rust-vmm` GitHub organization, plus one in-house crate. The versions pinned in `src/vmm/Cargo.toml` are:

Crate	Version	Role
`kvm-bindings`	0.14.0	FFI structs from `<linux/kvm.h>`: `kvm_run`, `kvm_regs`, `kvm_sregs`, `kvm_userspace_memory_region`, generated by rust-bindgen
`kvm-ioctls`	0.24.0	Safe wrappers: `Kvm`, `VmFd`, `VcpuFd`, `DeviceFd`, `KvmRunWrapper` (mmap of `kvm_run`)
`vm-memory`	0.17.1	`GuestAddress` GPA newtype, `GuestMemoryMmap`, `GuestRegionMmap`; dirty bitmap tracking via `backend-bitmap` feature
`vmm-sys-util`	0.15.0	`EventFd`, epoll wrappers, ioctl macro families
`linux-loader`	0.13.2	Loads `vmlinux` ELF on x86-64 and `Image` PE on aarch64; writes boot params to the zero page
`vm-allocator`	0.1.4	`AddressAllocator` for MMIO ranges; `IdAllocator` for device IDs
`vm-superio`	0.8.1	`Serial` (16550A UART), `I8042Device` (PS/2 / CPU reset), `Rtc` (PL031, aarch64 only)
`vhost`	0.15.0	vhost-user frontend for offloading virtio-net or virtio-fs to external backends
`vm-fdt`	0.3.0	Flattened Device Tree blob generation (aarch64 only)
`micro_http`	git	In-house HTTP/1.x over Unix socket; not published to crates.io

A few of these are worth examining in more detail because they encode design decisions that affect everything built on top of them.

kvm-ioctls and kvm-bindings

`kvm-bindings` is what you use when you need to pass a struct into a KVM ioctl: it provides the C types from `<linux/kvm.h>` as Rust FFI structs, generated by rust-bindgen from the kernel headers. `kvm-ioctls` wraps those into safe Rust: the `Kvm` struct wraps `/dev/kvm`; `VmFd` wraps the VM fd; `VcpuFd` wraps the vcpu fd. `VcpuFd::run()` is the `KVM_RUN` call. `VcpuFd::get_regs()` and `set_regs()` read and write `kvm_regs`; `get_sregs()` and `set_sregs()` handle `kvm_sregs`. `VmFd::register_irqfd()` and `register_ioeventfd()` wire up the kernel-side interrupt and I/O notification mechanisms without needing a VM exit.

vm-memory

`GuestAddress` is a newtype over `u64` representing a guest physical address (GPA). The separation matters: the host never accidentally uses a GPA as a host virtual address (HVA). `GuestMemoryMmap` is the concrete backed-by-`mmap` type; it holds a collection of `GuestRegionMmap` objects, each mapping a contiguous GPA range to a `MmapRegion` on the host. Cross-region reads and writes — a buffer that

straddles two memory regions — are handled transparently by the `GuestMemory` trait. The `backend-bitmap` Cargo feature (enabled by Firecracker) adds per-region dirty tracking; this is the userspace side of the dirty-page mechanism that `KVM_MEM_LOG_DIRTY_PAGES` enables in the kernel.

The Virtio Queue Firecracker Does Not Borrow

One notable absence from the crate list: Firecracker does not use the `virtio-queue` crate from rust-vmm. It reimplements the VIRTIO 1.2 split virtqueue in `src/vmm/src/devices/virtio/queue.rs`, with the copyright notice acknowledging both Amazon and the original Chromium OS authors. The internal `Queue` struct conforms to OASIS VIRTIO 1.2 section 2.7:

Part	Alignment	Size
Descriptor Table	16 bytes	16 × Queue Size
Available Ring	2 bytes	6 + 2 × Queue Size
Used Ring	4 bytes	6 + 8 × Queue Size

Queue Size must be a power of 2; the maximum is 32,768. The `virtq_desc` struct is 16 bytes: an `le64` guest-physical address, an `le32` length, an `le16` flags field (`VIRTQ_DESC_F_NEXT=1`, `VIRTQ_DESC_F_WRITE=2`, `VIRTQ_DESC_F_INDIRECT=4`), and an `le16` next-descriptor index. Firecracker's internal implementation includes formal verification with the Kani model checker and custom helpers — `prepare_kick()` and `try_enable_notification()` — for interrupt suppression. The decision to maintain a fork rather than adopt the shared crate keeps those guarantees internal and avoids taking a dependency on a crate whose API surface Firecracker does not fully control.

The VIRTIO MMIO transport register map sets the magic value `"virt"` at offset 0x000, version 2 at offset 0x004, and the `QueueNotify` kick register at offset 0x050.

The Pre-Boot / Running State Machine

A Firecracker process is either configuring a VM or running one. It cannot do both. This constraint is not enforced by convention; it is enforced by two distinct controller objects that replace each other at boot time.

The REST endpoint `GET /` returns an `InstanceInfo` struct whose `state` field is one of three strings: `"Not started"`, `"Running"`, or `"Paused"`. These states determine which operations are legal.

PrebootApiController

Before `InstanceStart`, the VMM thread handles requests through `PrebootApiController`. Every configuration mutation goes through this controller: `ConfigureBootSource`, `InsertBlockDevice`, `InsertNetworkDevice`, `InsertPmemDevice`, `PutCpuConfiguration`, `SetMmdsConfiguration`, `SetVsockDevice`, `SetEntropyDevice`, `SetBalloonDevice`, `UpdateMachineConfiguration`, and `LoadSnapshot`. The transition trigger is `StartMicroVm` (mapped to `PUT /actions` with `action_type: InstanceStart`). Once that action lands, `PrebootApiController` hands off to `RuntimeApiController` and is never consulted again.

The `MachineConfig` that the pre-boot controller accepts has these fields:

Field	Type	Default	Constraint
`vcpu_count`	u8	1	1–32; must be 1 or even if SMT enabled; SMT unsupported on aarch64
`mem_size_mib`	usize	128	> 0; must be a multiple of 2 if using 2 MiB huge pages; must be ≥ balloon target
`smt`	bool	false	x86-64 only
`track_dirty_pages`	bool	false	Required for diff snapshots
`huge_pages`	enum	None	`None` or `Hugetlbfs2M`

`track_dirty_pages: true` causes Firecracker to pass `KVM_MEM_LOG_DIRTY_PAGES` in the `flags` field of `kvm_userspace_memory_region` when registering DRAM slots. It cannot be toggled after boot.

RuntimeApiController

After `InstanceStart`, requests go to `RuntimeApiController`. The mutation operations available here are narrower: `Pause`, `Resume`, `CreateSnapshot`, `UpdateBlockDevice`, `UpdateNetworkInterface` (rate-limiter changes only), `UpdateBalloon`, `UpdateBalloonStatistics`, `SendCtrlAltDel` (x86-64 only), and a set of read operations: `GetBalloonStats`, `GetFullVmConfig`, `GetMMDS`, `PatchMMDS`.

Attempting a pre-boot-only action in this state returns `VmmActionError::OperationNotSupportedPostBoot`: "The requested operation is not supported after starting the microVM."

`CreateSnapshot` is only valid in `Paused` state. The caller first sends `PATCH /vm` with `{ "state": "Paused" }` — which stops returning to the event loop, blocking device emulation — and then `PUT /snapshot/create`. During pause, the vCPU deadlock detection timeout is 30 seconds (`RECV_TIMEOUT_SEC`). A `PATCH /vm` with `{ "state": "Resumed" }` lifts the pause.

Boot Sequence: builder.rs

The implementation of `StartMicroVm` lives in `src/vmm/src/builder.rs`, in `build_microvm_for_boot`. The sequence inside that function:

1. Validate that a kernel config is present (else `MissingKernelConfig` error).
2. `allocate_guest_memory()` — creates the `GuestMemoryMmap`, one region per DRAM slot.
3. `Kvm::new()` — opens `/dev/kvm`; create `KvmVm`; generate `VcpuFd` objects via `KVM_CREATE_VCPU`.
4. Register DRAM regions with KVM via `KVM_SET_USER_MEMORY_REGION`.
5. Create `DeviceManager`.
6. Load the kernel image via `linux-loader`. Firecracker requires an uncompressed `vmlinux` ELF on x86-64, not a `bzImage`. On x86-64, `linux-loader` uses the 64-bit protected-mode entry point directly, bypassing the real-mode decompressor entirely. On aarch64, it loads a `Image` PE.
7. Attach devices in order: boot timer, balloon, block (root first), network, pmem, vsock, entropy/RNG, virtio-mem hotplug, legacy devices (aarch64 only), VMGenID, VMClock.
8. `configure_system_for_boot()` — architecture-specific: on x86-64, patches the kernel command line and zero page; on aarch64, constructs the Flattened Device Tree via `vm-fdt`. Firecracker uses MMIO throughout — there is no PCI bus on either architecture — so no PCI configuration is performed here.
9. Spawn vCPU threads via `KvmVm::start_vcpus()`. Each thread applies its seccomp-BPF filter before doing any other work; a missing `"vcpu"` entry in the `BpfThreadMap` panics the thread with `MissingSeccompFilters`.
0. Register the `Vmm` struct with `EventManager`.
11. The VMM is now in the **Paused** state. `Vmm::resume_vm()` signals every vCPU thread to enter `KVM_RUN`.

Step 6 is where the uncompressed `vmlinux` requirement bites developers who try to use a `bzImage` directly: the ELF header check in `linux-loader` fails fast, rather than silently executing the real-mode decompressor stub and producing a boot that hangs without explanation.

```
sequenceDiagram
  participant API as "API thread (fc_api)"
  participant VMM as "VMM thread"
  participant KVM as "/dev/kvm"
  participant vCPU as "fc_vcpu 0"
  participant Guest as "Guest kernel"

  API->>VMM: InstanceStart (via mpsc + EventFd)
  VMM->>KVM: "KVM_CREATE_VM → VmFd"
  VMM->>KVM: "KVM_SET_USER_MEMORY_REGION (DRAM slots)"
  VMM->>KVM: "KVM_CREATE_VCPU → VcpuFd"
  VMM->>vCPU: spawn fc_vcpu 0 (paused)
  VMM->>VMM: attach devices, configure system
  VMM->>vCPU: resume_vm() (VcpuEvent::Resume)
  vCPU->>KVM: "KVM_RUN"
  KVM-->>vCPU: "VM exit: KVM_EXIT_MMIO (virtio probe)"
  vCPU->>vCPU: "mmio_bus.read/write (inline, no VMM wake)"
  vCPU->>KVM: "KVM_RUN (re-enter)"
  KVM-->>Guest: guest executes at kernel entry
  Guest->>Guest: virtio device enumeration
  Guest->>Guest: "kernel_init execve /sbin/init"
```

The diagram above condenses the loop: after the first MMIO exit, the vCPU continues issuing `KVM_RUN` calls and handling exits until the guest is fully initialized. Each MMIO exit is handled inline in the vCPU thread — `handle_kvm_exit()` calls `mmio_bus.read()` or `mmio_bus.write()` directly on the `Peripherals` struct attached to that vCPU. The mpsc channel to the VMM thread carries only control events (Pause, Resume, SaveState).

The Seccomp Boundary

Each of the three thread types receives a distinct seccomp-BPF filter, applied inside the spawned thread before any other work. The filter map is keyed by the strings `"api"`, `"vmm"`, and `"vcpu"`. Firecracker's design document states the threat model directly: "All vCPU threads are considered to be running malicious code as soon as they have been started; these malicious threads need to be contained." The vCPU threads have the narrowest allowlist for exactly this reason: they make a `KVM_RUN` ioctl, read from `kvm_run`, and do almost nothing else. The API thread and VMM thread need a wider set — socket operations, file I/O for block devices, `epoll_wait` — but each is still a small, auditable list.

The filters are compiled at build time from JSON by `seccompiler-bin` and embedded in the binary. A failure to install the filter in any thread class causes that thread to panic; the panic propagates through the `exit_evt` `EventFd` to the VMM's `EventManager`, which shuts the process down cleanly. There is no path through which a misconfigured filter silently degrades to permissive mode.

Firecracker's design document describes a concentric trust model, nesting several zones from least trusted (guest vCPU threads) to most trusted (host). The hardware VMX boundary separates guest from VMM; the seccomp filter separates the VMM from most of the host kernel syscall table; the jailer's chroot and namespaces separate the Firecracker process from the rest of the host filesystem. Chapter 18 covers the jailer in detail; Chapter 19 covers the seccomp filters.

What One firecracker Process Holds Open

After a guest is started, a single `firecracker` process holds:

- `/dev/kvm` (system fd, obtained via `Kvm::new()`)
- One VM fd (from `KVM_CREATE_VM`)
- One vcpu fd per guest core (from `KVM_CREATE_VCPU`), each with a `mmap`'d `kvm_run` region
- The Unix socket fd for the API (`/run/firecracker.socket`)
- One `EventFd` per device for virtio queue notifications (the `ioeventfd`s registered with `VmFd::register_ioeventfd()`)
- One `EventFd` per interrupt for guest interrupt injection (the `irqfd`s registered with `VmFd::register_irqfd()`)
- One tap fd per `virtio-net` device (the host-side network interface)
- One file fd per `virtio-block` device (the rootfs or data image)
- The `EventFd` for the API-to-VMM wakeup
- The `EventFd` per vCPU for exit signaling

The total number of file descriptors scales with the device count. A minimal guest — one vCPU, one block device, one network interface — holds on the order of two dozen fds. The VMM memory overhead at this configuration, excluding guest RAM, is specified in `SPECIFICATION.md` as no more than 5 MiB; in practice it runs around 3 MiB.

That ceiling, 5 MiB of overhead for a complete VMM process, is what makes packing thousands of microVMs onto a single host tractable. The Firecracker USENIX ATC 2020 paper reports QEMU's VMM overhead at roughly 131 MiB per VM (Table 1 of the paper); against Firecracker's 5 MiB ceiling the gap is more than 25×. The difference comes from the minimal device model and the absence of a general-purpose emulation layer — the next chapter opens on the device set that stays within that envelope.

Sources And Further Reading

- Agache et al., "Firecracker: Lightweight Virtualization for Serverless Applications," USENIX ATC 2020: https://www.usenix.org/conference/atc20/presentation/agache (Table 1 VMM overhead comparison)
- Firecracker design document: https://github.com/firecracker-microvm/firecracker/blob/main/docs/design.md

- Firecracker `SPECIFICATION.md`: https://github.com/firecracker-microvm/firecracker/blob/main/SPECIFICATION.md
- Firecracker getting-started guide: https://github.com/firecracker-microvm/firecracker/blob/main/docs/getting-started.md
- Firecracker jailer documentation: https://github.com/firecracker-microvm/firecracker/blob/main/docs/jailer.md
- Firecracker snapshotting documentation: https://github.com/firecracker-microvm/firecracker/blob/main/docs/snapshotting/snapshot-support.md
- Firecracker actions API documentation: https://github.com/firecracker-microvm/firecracker/blob/main/docs/api_requests/actions.md
- OpenAPI specification (`firecracker.yaml`): https://raw.githubusercontent.com/firecracker-microvm/firecracker/main/src/firecracker/swagger/firecracker.yaml
- `src/vmm/Cargo.toml` (crate version pins): https://raw.githubusercontent.com/firecracker-microvm/firecracker/main/src/vmm/Cargo.toml
- `src/firecracker/src/api_server_adapter.rs` (`fc_api` thread spawn and `ApiServer::new`): https://raw.githubusercontent.com/firecracker-microvm/firecracker/main/src/firecracker/src/api_server_adapter.rs
- `src/firecracker/src/api_server/mod.rs` (payload size limit, HTTP server loop): https://raw.githubusercontent.com/firecracker-microvm/firecracker/main/src/firecracker/src/api_server/mod.rs
- `src/vmm/src/vstate/vcpu.rs` (vCPU thread spawn, `KVM_RUN` loop, exit handler): https://raw.githubusercontent.com/firecracker-microvm/firecracker/main/src/vmm/src/vstate/vcpu.rs
- `src/vmm/src/vstate/vm.rs` (`KVM_CREATE_VCPU`, `start_vcpus`): https://raw.githubusercontent.com/firecracker-microvm/firecracker/main/src/vmm/src/vstate/vm.rs
- `src/vmm/src/builder.rs` (`build_microvm_for_boot`, `EventManager` registration): https://raw.githubusercontent.com/firecracker-microvm/firecracker/main/src/vmm/src/builder.rs
- `src/vmm/src/rpc_interface.rs` (`PrebootApiController`, `RuntimeApiController`, `VmmAction` enum): https://raw.githubusercontent.com/firecracker-microvm/firecracker/main/src/vmm/src/rpc_interface.rs
- `src/vmm/src/vmm_config/machine_config.rs` (`MachineConfig` fields and constraints): https://raw.githubusercontent.com/firecracker-microvm/firecracker/main/src/vmm/src/vmm_config/machine_config.rs
- `src/vmm/src/devices/virtio/queue.rs` (internal virtqueue, not using `virtio-queue` crate): https://raw.githubusercontent.com/firecracker-microvm/firecracker/main/src/vmm/src/devices/virtio/queue.rs
- `micro-http` crate (in-house HTTP/1.x server): https://github.com/firecracker-microvm/micro-http
- `kvm-ioctls` crate documentation: https://docs.rs/kvm-ioctls/latest/kvm_ioctls/

- `kvm-bindings` crate documentation: https://docs.rs/kvm-bindings/latest/kvm_bindings/
- `vm-memory` crate documentation and design document: https://docs.rs/vm-memory/latest/vm_memory/ and https://github.com/rust-vmm/vm-memory/blob/main/DESIGN.md
- `vmm-sys-util` crate documentation: https://docs.rs/vmm-sys-util/latest/vmm_sys_util/
- `linux-loader` crate documentation: https://docs.rs/linux-loader/latest/linux_loader/
- `vm-superio` crate documentation: https://docs.rs/vm-superio/latest/vm_superio/
- `vm-allocator` crate documentation: https://docs.rs/vm-allocator/latest/vm_allocator/
- `vhost` crate documentation: https://docs.rs/vhost/latest/vhost/
- OASIS VIRTIO 1.2 specification (split virtqueue layout §2.7, MMIO transport §4.2): https://docs.oasis-open.org/virtio/virtio/v1.2/csd01/virtio-v1.2-csd01.html
- Linux KVM API documentation: https://docs.kernel.org/virt/kvm/api.html
- Linux `<linux/kvm.h>` (ioctl numbers, `kvm_run` exit reasons): https://raw.githubusercontent.com/torvalds/linux/master/include/uapi/linux/kvm.h
- Linux x86 boot protocol: https://docs.kernel.org/arch/x86/boot.html

Chapter 14: Firecracker's Device Model

If you run `qemu-system-x86_64 --help` and scroll through the device list, you will count hundreds of options: ISA bridges, SCSI controllers, USB hubs, VGA cards, audio codecs, NVMe controllers, PCI root complexes, and dozens more. A real cloud provider running QEMU must audit and seccomp-filter every one of those paths, because a compromised guest can reach any of them through the device protocol. The question Firecracker was designed to answer is whether a general-purpose virtual machine actually needs all of that surface, or whether a server workload can live comfortably inside a much shorter list.

The answer Firecracker gives in production is: seven device types. That is not a capability gap or a temporary limitation. It is the security model made concrete.

The Device Inventory

Firecracker's `FAQ.md` states the set directly: virtio-net, virtio-balloon, virtio-block, virtio-vsock, serial console, and a minimal keyboard controller. A seventh device — virtio-rng — was added in v1.4.0 and is exposed through the `/entropy` API endpoint. Every virtio device uses the MMIO transport; there is no PCI bus, no PCIe root complex, no USB controller, no GPU, no audio subsystem, no NVMe or SCSI stack, no virtio-console (device type 3), no virtio-GPU (type 16), no virtio-fs (type 26), no virtio-input (type 18), and no virtio-crypto (type 20). There is also no ACPI power management plane.

The full inventory as of Firecracker `main` (June 2026):

Device	Virtio ID	Transport	Notes
virtio-net	1	MMIO	RX + TX queues; MMDS intercept embedded here
virtio-block	2	MMIO	1 queue; sync or io_uring file engine
virtio-rng	4	MMIO	1 queue; entropy from aws-lc-rs
virtio-balloon	5	MMIO	Traditional memory balloon
virtio-vsock	19	MMIO	3 queues; AF_UNIX backend on host
Serial console	—	I/O port 0x3F8, IRQ 4	16550A UART via vm-superio
i8042	—	I/O port 0x060, IRQ 1	Reset signalling only

The virtio IDs come from `src/vmm/src/devices/virtio/generated/virtio_ids.rs`; the legacy device registrations from `src/vmm/src/device_manager/legacy.rs`.

The two non-virtio entries — the serial console and the i8042 keyboard controller — deserve a word each. The i8042 is not there to support a keyboard. Its sole function is to let the guest signal a reboot. `docs/design.md` says as much: "Within Firecracker, the purpose of the I8042 device is to signal the

microVM that the guest has requested a reboot." It is registered at I/O port `0x060`, 5 bytes, IRQ GSI 1. Because the guest kernel is told via boot parameters (`i8042.noaux i8042.nomux i8042.dumbkbd`) not to probe the controller's hardware features, the actual I/O surface is small and the interaction is deterministic.

The serial console is a 16550A-compatible UART emulated by the `vm-superio` crate, specifically `vm_superio::Serial`. On x86_64 it appears at I/O port `0x3F8` (COM1), size 8 bytes, IRQ GSI 4. On aarch64 the same serial device is memory-mapped at `SERIAL_MEM_START` rather than an I/O port — a platform difference the guest kernel handles through its own driver but that the chapter source code in `legacy.rs` makes explicit per-arch. The serial console is not a virtio device and does not appear in the MMIO device region; it is a classic ISA-style I/O-port device that predates the virtio era entirely.

The virtio-MMIO Transport

Connecting the guest to a device requires a transport: a shared protocol by which the guest driver and the host device negotiate capabilities, exchange buffers, and signal completion. QEMU supports two transports for virtio: PCI and MMIO. Firecracker uses only MMIO, implementing OASIS virtio specification version 1.2, §4.2.

The choice matters for boot latency and attack surface simultaneously. A PCI bus requires an ACPI table for device discovery, a host bridge emulator, interrupt routing through an I/O APIC, and MSI/MSI-X plumbing for modern devices. MMIO skips all of that: the guest learns about each device's location from the kernel command line (the `virtio_mmio.device=` parameter), and the transport consists of a fixed-layout register window per device. There is nothing dynamic to probe and nothing to enumerate.

Each virtio-MMIO device in Firecracker occupies exactly `MMIO_LEN = 0x1000` bytes (4 KiB) of MMIO address space. On x86_64 the first device slot starts at `MEM_32BIT_DEVICES_START = 0xC000_1000`, immediately after a boot timer device that sits at `MMIO32_MEM_START = 0xC000_0000`. Each additional device is assigned the next 4 KiB-aligned slot via `AllocPolicy::FirstMatch` in `src/vmm/src/device_manager/mmio.rs`. The address layout is fixed at VM construction time; there is no hotplug.

The Register Map

Every virtio-MMIO device exposes the same register layout at the base of its 4 KiB window, defined by virtio 1.2 §4.2.2 and implemented in `src/vmm/src/devices/virtio/transport/mmio.rs`:

Offset	Size	R/W	Field
0x000	4	RO	MagicValue (`0x7472_6976`, ASCII "virt")
0x004	4	RO	Version (`0x2`; not legacy `0x1`)
0x008	4	RO	DeviceID (virtio type integer)
0x00c	4	RO	VendorID (`0x0`)
0x010	4	RO	DeviceFeatures (page selected by 0x014)
0x014	4	WO	DeviceFeaturesSel
0x020	4	WO	DriverFeatures
0x024	4	WO	DriverFeaturesSel
0x030	4	WO	QueueSel
0x034	4	RO	QueueNumMax (queue's `max_size`)
0x038	4	WO	QueueNum
0x044	4	RW	QueueReady
0x050	4	WO	QueueNotify (`NOTIFY_REG_OFFSET`)
0x060	4	RO	InterruptStatus (bit 0 = vring, bit 1 = config change)
0x064	4	WO	InterruptACK
0x070	4	RW	DeviceStatus
0x080	4	WO	QueueDescLow
0x084	4	WO	QueueDescHigh
0x090	4	WO	QueueAvailLow
0x094	4	WO	QueueAvailHigh
0x0a0	4	WO	QueueUsedLow
0x0a4	4	WO	QueueUsedHigh
0x0fc	4	RO	ConfigGeneration
0x100–0xfff	var	RW	Device-specific configuration space

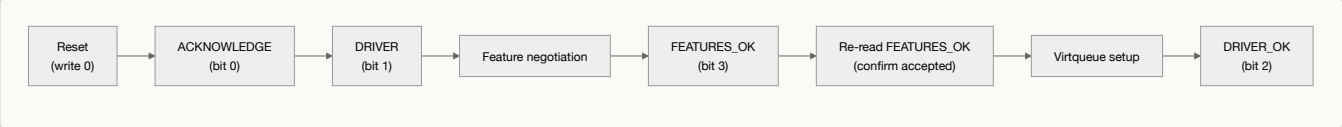
The MagicValue at offset 0x000 reads `0x7472_6976` — the ASCII bytes for "virt" stored little-endian. A guest driver that sees any other value knows immediately that the MMIO window does not contain a virtio device. The Version field reads `0x2`, explicitly distinguishing this from the legacy virtio 0.9 transport

(which reads `0x1`); Firecracker does not support the legacy transport.

Initialization: The Status State Machine

Device initialization follows a strict state machine mandated by virtio 1.2 §3.1.1 and enforced in `src/vmm/src/devices/virtio/mod.rs`. The sequence is: write 0 to DeviceStatus (reset), then set `ACKNOWLEDGE` (bit 0, value 1) to signal the driver has found the device, then set `DRIVER` (bit 1, value 2) to signal it knows how to drive it. After that comes feature negotiation — the driver reads DeviceFeatures with DeviceFeaturesSel cycling through 32-bit pages, sets DriverFeatures similarly, then sets `FEATURES_OK` (bit 3, value 8) and re-reads DeviceStatus to confirm the device accepted the negotiated feature set. Only if FEATURES_OK is still set does the driver proceed to configure virtqueues and finally set `DRIVER_OK` (bit 2, value 4) to indicate readiness.

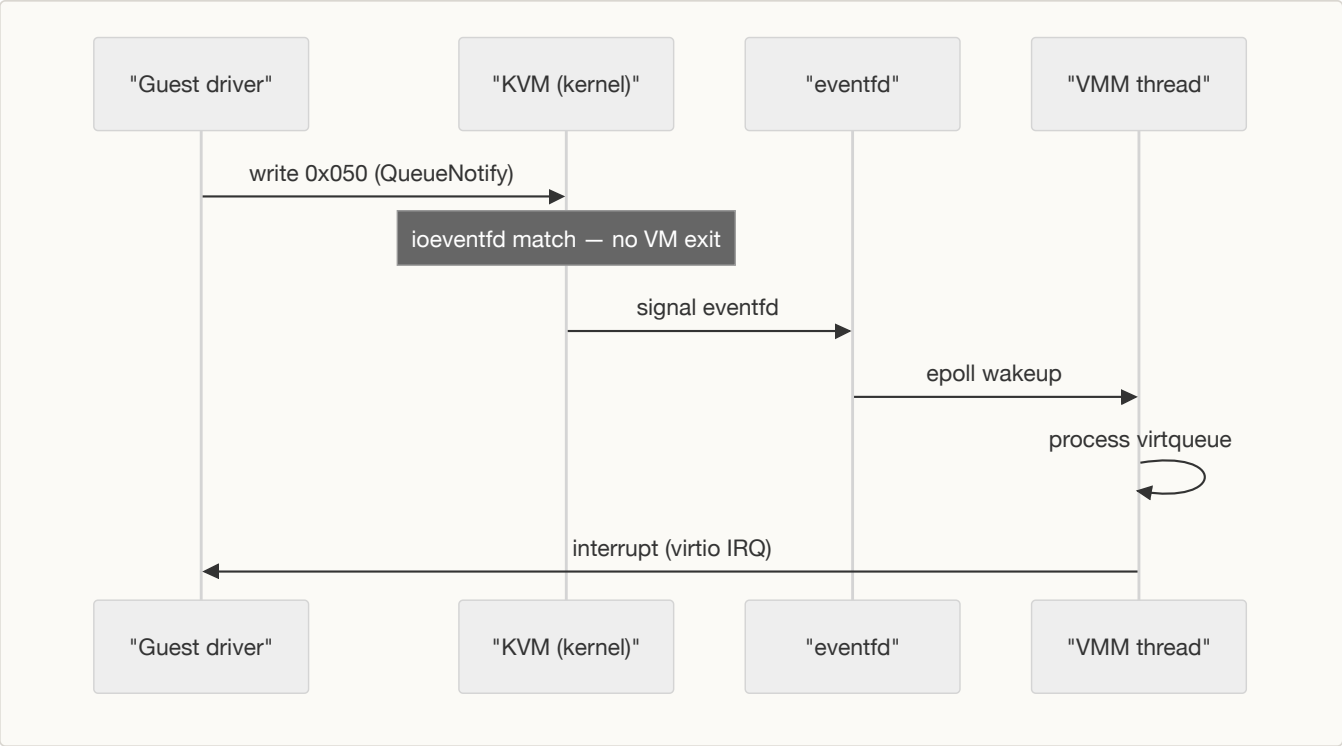
Two additional bits signal abnormal states: `DEVICE_NEEDS_RESET` (bit 6, value 64) is set by the device when it encounters an unrecoverable error; `FAILED` (bit 7, value 128) is set by the driver when it gives up. Writes that set FAILED are accepted at any point; clearing individual bits other than a full reset (writing 0) is rejected. The state machine is not a suggestion — a guest that attempts to skip steps gets a device that stays inert.



Queue Notification and ioeventfd

The most performance-sensitive operation in the virtio protocol is the queue notification: the moment the guest driver tells the host that it has placed new buffers into the available ring. If this notification required a full VM exit — the vCPU exiting to kernel mode, the kernel dispatching the MMIO write to the VMM, the VMM processing it — the round-trip cost would dominate I/O latency.

Firecracker avoids this via KVM's `ioeventfd` mechanism. When a device is registered, the VMM calls `KVM_IOEVENTFD` to install an `ioeventfd` at the device's notify register address, `device_base + NOTIFY_REG_OFFSET` where `NOTIFY_REG_OFFSET = 0x050`. The queue index is passed as the datamatch value. From that point on, the guest's write to offset 0x050 causes KVM to signal the corresponding `eventfd` file descriptor directly, without returning to userspace through the normal MMIO exit path. The VMM thread watching that eventfd wakes up and processes the queue — still in userspace, still in the `firecracker` process, but without a kernel round-trip for the notification itself. The code is in `src/vmm/src/device_manager/mmio.rs`.



The IRQ Limit

Originally Firecracker gave each virtio device its own legacy GSI line, which capped the device count at 11. IRQ sharing was added in PR #2286 to relax that limit. The current ceiling is 19 virtio devices, bounded by available legacy GSI lines (GSI 5 through 23) rather than by MMIO address space or any architectural constraint. In practice, the defined device catalog reaches nowhere near that ceiling — the constraint is policy, not capacity.

The Individual Devices

virtio-net

The network device has two queues: RX (index 0) and TX (index 1). There is no control virtqueue. Both queues are capped at `NET_QUEUE_MAX_SIZE = 256` descriptors. The maximum frame buffer size is `MAX_BUFFER_SIZE = 65562` bytes. The constant is defined as the raw integer in `src/vmm/src/devices/virtio/net/mod.rs` without an explanatory comment; the arithmetic that accounts for it — 65536-byte maximum Ethernet payload, 12-byte `virtio_net_hdr_v1` header, 14-byte Ethernet frame header — is inferred from the value.

The feature bits Firecracker always advertises include checksum offload on both sides (`VIRTIO_NET_F_CSUM` bit 0, `VIRTIO_NET_F_GUEST_CSUM` bit 1), TSO4 and TSO6 in both directions (bits 7, 8, 11, 12), UFO in both directions (bits 10, 14), receive buffer merging (VIRTIO_NET_F_MRG_RXBUF bit 15), VIRTIO_F_VERSION_1 (bit 32), and VIRTIO_RING_F_EVENT_IDX (bit 29). MAC address support (VIRTIO_NET_F_MAC bit

5) and MTU advertisement (VIRTIO_NET_F_MTU bit 3) are offered conditionally when the respective values are configured at VM creation time.
Notably, VIRTIO_NET_F_STATUS (bit 16) and multiqueue support VIRTIO_NET_F_MQ` (bit 22) are intentionally not advertised — there is no live link status signalling and only one pair of queues.

The device-specific configuration space exposed at register offset 0x100 holds: `guest_mac` (6 bytes), a status word and a max virtqueue pairs word that are both present for protocol compliance but unused, and the MTU (2 bytes).

virtio-block

The block device uses a single queue with a maximum of 256 descriptors (`FIRECRACKER_MAX_QUEUE_SIZE = 256`) and addresses storage in 512-byte sectors. The feature set is pared to what a container workload actually needs: `VIRTIO_F_VERSION_1` (bit 32) and `VIRTIO_RING_F_EVENT_IDX` (bit 29) always; `VIRTIO_BLK_F_FLUSH` (bit 9) when the drive is configured with writeback caching; `VIRTIO_BLK_F_RO` (bit 5) for read-only drives. Geometry, topology, and size/segment-max negotiation are not advertised. The device configuration space contains exactly one field: `capacity` , a `u64` counting 512-byte sectors.

Firecracker implements three operation types: `VIRTIO_BLK_T_IN` (read), `VIRTIO_BLK_T_OUT` (write), and `VIRTIO_BLK_T_FLUSH` . Discard (`VIRTIO_BLK_T_DISCARD`) and write-zeroes (`VIRTIO_BLK_T_WRITE_ZEROES`) are not implemented — the tradeoff is explicit in the source rather than simply absent.

The device supports two file engines: a synchronous engine that issues blocking syscalls, and an asynchronous engine backed by io_uring with a submission queue depth of `IO_URING_NUM_ENTRIES = 128` entries. Disk identity for snapshot versioning is a 20-byte ASCII string (`VIRTIO_BLK_ID_BYTES = 20`) derived from the backing file's `st_dev` , `st_rdev` , and `st_ino` fields.

virtio-vsock

The vsock device provides a stream socket channel between the guest and a process on the host without requiring a network interface or IP routing. It has three queues: RX (index 0), TX (index 1), and Event (index 2), each sized 256 descriptors. Feature bits: `VIRTIO_F_VERSION_1` (bit 32), `VIRTIO_F_IN_ORDER` (bit 35), and `VIRTIO_RING_F_EVENT_IDX` (bit 29). `VIRTIO_VSOCK_F_SEQPACKET` is not advertised.

The host CID is hardcoded as `VSOCK_HOST_CID = 2` , as the virtio-vsock specification reserves. The guest CID is user-configured at VM creation time.

The backend is an AF_UNIX socket on the host, not the `vhost-vsock` kernel module. This is deliberate. vhost would move the data path into the kernel, eliminating some context switches, but the kernel would then become directly reachable from the guest — which is precisely the attack surface that the combination of KVM and seccomp filtering is designed to interpose. From Issue #650: "we don't want to

use vhost since that would be another attack surface to directly expose the host kernel." The AF_UNIX backend stays in the `firecracker` userspace process, subject to the same seccomp filter as every other VMM operation.

Connection routing works as follows: connections that the host initiates arrive at the single UDS path configured at boot time; Firecracker forwards them into the guest on the appropriate port. Connections that the guest initiates cause Firecracker to open a UDS path derived from the configured path plus the target port number — for example, if the UDS path is `./v.sock`, a guest connection to port 52 causes Firecracker to connect to `./v.sock_52`. The maximum per-packet buffer is `MAX_PKT_BUF_SIZE = 64 * 1024` (64 KiB).

virtio-rng

The entropy device is the newest addition to the standard catalog, shipping in Firecracker v1.4.0. It has one queue, 256 descriptors, and no device-specific configuration space — `read_config` and `write_config` are both no-ops, which is correct: OASIS virtio 1.2 §5.4 defines no device-specific feature bits or configuration fields for RNG devices. The only feature bit is `VIRTIO_F_VERSION_1` (bit 32).

The entropy source is `aws-lc-rs`, AWS's Rust bindings to AWS-LC — a FIPS-validated fork of BoringSSL. Each request calls `aws_lc_rs::rand::fill(&mut rand_bytes)`. Individual requests are capped at `MAX_ENTROPY_BYTES = 64 * 1024` (65536 bytes) to prevent a single oversized descriptor chain from monopolizing the device. On the guest side the device appears as `/dev/hwrng`, which requires `CONFIG_HW_RANDOM_VIRTIO=y` in the kernel configuration.

The rationale for using `aws-lc-rs` rather than the kernel's `getrandom(2)` comes down to the same seccomp concern seen elsewhere in Firecracker: `getrandom` is not in the Firecracker seccomp allowlist by default, and relying on a library that manages its own entropy pool avoids the need to add a system call to the filter surface.

virtio-balloon

The balloon device appears in the inventory table but is not covered in depth here — its interface is straightforward enough not to warrant it. It exposes two queues: an inflate queue (index 0) on which the guest returns physical page frame numbers the host may reclaim, and a deflate queue (index 1) on which the host returns pages to the guest. A third stats queue is defined by the virtio spec but is optional and not all configurations enable it. The balloon device has no device-specific configuration fields beyond the standard feature negotiation. Its role in a microVM context is memory overcommit recovery — allowing the host to reclaim guest pages under pressure without a full guest shutdown.

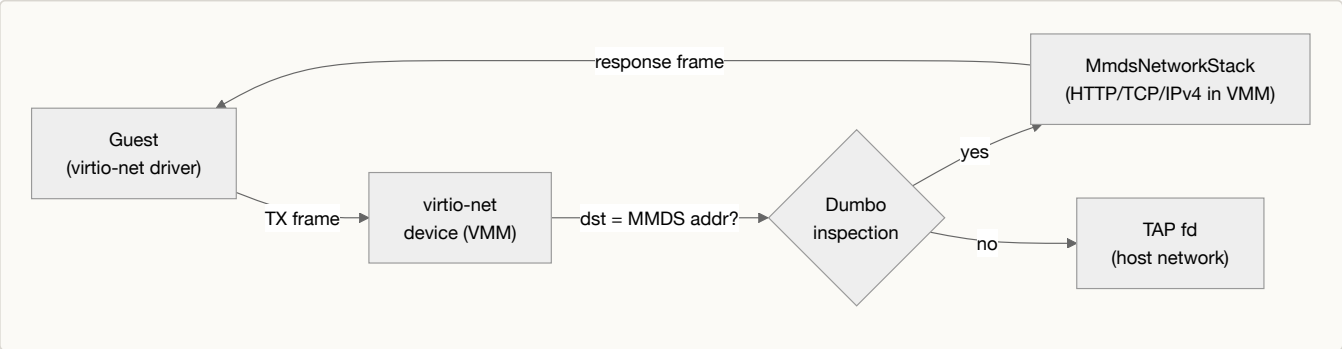
MMDS: Metadata Without a Second Network Interface

Every cloud provider needs a way to deliver instance metadata to a running virtual machine — the AWS instance identity document, credentials refreshed from IAM, user data scripts, and similar dynamic configuration. The traditional approach is to run a separate HTTP server on a link-local address

(`169.254.169.254` in the AWS case) and route guest traffic to it. For a microVM that starts in well under a second and runs for seconds, standing up a separate network path for metadata has disproportionate cost.

Firecracker's solution is MMDS — the MicroVM Metadata Service — implemented not as a separate device but as a software shim embedded inside the virtio-net device path. The implementation is called `MmdsNetworkStack`, or "Dumbo" in internal Firecracker naming. Every Ethernet frame leaving the guest through the virtio-net TX queue is inspected before being forwarded to the host TAP file descriptor. If the frame's IPv4 destination address matches the configured MMDS address, it is diverted to Dumbo instead of transmitted.

Dumbo is a bespoke HTTP 1.1 / TCP / IPv4 stack implemented inside the `firecracker` process, outside the KVM boundary. It is deliberately narrow: it handles GET and PUT requests only, implements no congestion control, handles no 802.1Q VLAN tagging, and does no IP fragmentation. ARP responses use the fixed MAC address `06:01:23:45:67:01`. The implementation lives in `src/vmm/src/devices/virtio/net/device.rs`, woven into the same code that handles normal frame forwarding.



MMDS supports two versions. V1 requires no authentication — any guest process can read metadata — and is deprecated. V2 follows the IMDSv2 pattern: the guest first sends a PUT to `/latest/api/token` with the header `X-metadata-token-ttl-seconds` (range: 1 through 21600 seconds), receives a token, and presents that token in subsequent GET requests via `X-metadata-token` or `X-aws-ec2-metadata-token`. This prevents server-side request forgery attacks where a guest web server is tricked into relaying metadata to an external attacker.

The elegance of MMDS is that it adds no device, no IRQ, no MMIO window, no additional seccomp filter surface, and no process boundary. It reuses the packet processing path that is already active for every frame the guest sends.

Rate Limiters

Running multi-tenant workloads on shared hardware requires more than memory and CPU isolation. A guest that drives its virtio-block device at full disk bandwidth starves every other microVM on the same host. Firecracker addresses this with token-bucket rate limiters that can be applied per-device to two independent dimensions: bytes (bandwidth) and operations per second (ops).

Rate limiting for block and net devices was introduced in Firecracker v0.4.0. The one-time burst feature was added in v0.7.0. The ability to update rate limiters on running block and net devices via `PATCH` was added in v0.24.0. As of v1.16.0, rate limiting was extended to the serial console (PR #5824).

The Token Bucket

The implementation lives in `src/vmm/src/rate_limiter/mod.rs` as `TokenBucket`:

```
pub struct TokenBucket {
    size: u64,                    // max capacity (tokens)
    initial_one_time_burst: u64,  // original burst budget, never replenished
    refill_time: u64,             // ms to refill from 0 to size
    one_time_burst: u64,          // remaining burst credit
    budget: u64,                  // current token budget
    last_update: Instant,
    processed_capacity: u64,      // size / gcd(size, refill_time_ns)
    processed_refill_time: u64,   // refill_time_ns / gcd(size, refill_time_ns)
}
```

`TokenBucket::new()` returns `None` — rate limiting disabled — when `size == 0` or `complete_refill_time_ms == 0`. An enabled bucket starts full (`budget == size`). Refill is computed on demand rather than on a tick:

```
refill_amount = (time_delta_ns * processed_capacity) / processed_refill_time
```

The fields `processed_capacity` and `processed_refill_time` are the GCD-reduced forms of `size` and `refill_time_ns`. This prevents integer overflow in the multiplication without losing precision on the refill rate. Sub-token time accumulation is handled by advancing `last_update` by the exact nanoseconds consumed for the tokens generated, not by the raw elapsed time — so fractional-token time carries forward and the long-run rate is exact.

The `reduce()` Call and Its Three Outcomes

When a device wants to process a request, it calls `TokenBucket::reduce(n)` where `n` is the token cost of the operation. The call returns one of three variants:

`BucketReduction::Success` — the budget covered the request; tokens are deducted and processing proceeds immediately.

`BucketReduction::Failure` — even after a passive replenish (update `budget` based on elapsed time since `last_update`), there are not enough tokens. The request is deferred.

`BucketReduction::OverConsumption(f64)` — the request exceeds the bucket's total `size`, meaning it would need to borrow more than one full refill cycle's worth of tokens. This case resolves Issue #259, which identified a deadlock: if the only request in flight is larger than the bucket size, a strict "must have tokens" policy blocks it forever, because the bucket can never grow large enough. Instead, Firecracker allows it, drains `budget` to zero, and returns the overconsumption ratio as a `f64`. The comment in `mod.rs` describes it as "overconsumption of (remaining tokens / size) times larger than the bucket size"; the formula is `(request_size - remaining_budget) / bucket_size`, where `remaining_budget` is what is left after passive auto-replenishment at the moment of the call. The ratio is used to compute an appropriate back-pressure delay.

The `one_time_burst` field is consumed first. If the burst allowance covers the entire request, the regular `budget` is not touched at all. This means a freshly-started microVM can absorb a startup I/O spike beyond the steady-state rate limit before the regular bucket begins draining.

The RateLimiter and Its Timer

```
pub struct RateLimiter {
    bandwidth: Option<TokenBucket>,   // byte tokens
    ops: Option<TokenBucket>,         // operation tokens
    timer_fd: TimerFd,
    timer_active: bool,
}
```

A single `RateLimiter` pairs two independent `TokenBucket` instances — one for bandwidth (bytes), one for operation count — with a `TimerFd` for deferred resumption. Both buckets must agree that a request can proceed; if either returns `Failure`, the timer is armed for `REFILL_TIMER_DURATION = Duration::from_millis(100)` (a compile-time constant, not configurable via API). On `OverConsumption(ratio)`, the timer is armed for `ratio * refill_time` milliseconds instead. While `timer_active == true`, `consume()` returns `false` immediately, gating both buckets with a single timer regardless of which one triggered the throttle.

There is a subtle implementation point worth naming: the `TimerFd` is created at `RateLimiter::new()` time even when both buckets are `None` and rate limiting is effectively disabled. The reason is that the Firecracker seccomp filter may block `timerfd_create(2)` at the time a rate limiter is later re-enabled via `update_buckets()`. Creating the fd up front, when the process still has an unrestricted syscall surface, prevents a later seccomp violation from silently disabling rate limiting at runtime.

`RateLimiter` implements `AsRawFd`. The VMM's epoll loop watches the timer fd and calls `event_handler()` when it fires, which clears `timer_active` and allows the device's queue processing to resume.

API Schema and Runtime Updates

The `TokenBucket` object in Firecracker's OpenAPI spec (`src/firecracker/swagger/firecracker.yaml`) exposes three fields: `size` (int64, minimum 0 — total token capacity), `refill_time` (int64, in milliseconds — time to refill from empty to full), and `one_time_burst` (int64, optional). The effective steady-state rate in tokens per second is `size / (refill_time / 1000)`.

A `RateLimiter` object wraps two optional `TokenBucket` instances under the keys `bandwidth` and `ops`. Rate limiters are accepted on:

- `PUT /drives/{drive_id}` and `PATCH /drives/{drive_id}` (block devices, ops and bandwidth)
- `PUT /network-interfaces/{iface_id}` and `PATCH /network-interfaces/{iface_id}` (net, ops and bandwidth)
- `PUT /serial` (serial console, since v1.16.0)
- `PUT /pmem/{id}` (virtio-pmem, since v1.16.0, PR #5789)

virtio-pmem is not part of the standard device catalog described in this chapter, but it accepts the same `RateLimiter` schema as the devices above.

To disable an existing rate limiter while the VM is running, send `size: 0, refill_time: 0`; `TokenBucket::new()` returns `None` and `update_buckets()` sets the field to `None`, removing the throttle entirely until a new non-zero configuration is applied.

Rate limiter state survives snapshot-restore. The persisted form `TokenBucketState` in `src/vmm/src/rate_limiter/persist.rs` stores `size`, `one_time_burst`, `refill_time`, the current `budget`, and `elapsed_ns` (nanoseconds since `last_update`). Restoring this state reconstructs the bucket mid-flight, preserving the throttle window that was active at snapshot time rather than resetting every bucket to full.

Why the List Is Short

The device model's length is not an engineering tradeoff between development time and capability. It is a direct consequence of the threat model stated in `docs/design.md`:

> *"All vCPU threads are considered to be running malicious code as soon as they have been started; these malicious threads need to be contained."*

Every device emulator is code reachable from the guest through the virtio queue protocol — or, for legacy devices, through I/O port reads and writes. A device class that does not exist cannot have an exploitable implementation bug. QEMU's device catalog has been the source of dozens of CVEs: heap overflows in the Cirrus VGA emulator, out-of-bounds writes in the ATI MMIO handler, USB subsystem escapes, NVMe controller vulnerabilities. Firecracker avoids that surface by never implementing those device classes, not by attempting to audit and harden them.

The vsock backend makes the same argument — see §virtio-vsock — and so does the choice of MMIO over PCI: narrower transport, smaller guest-reachable surface. The principle generalizes across every entry in the inventory.

The question for each potential addition is not "could we implement this?" but "does this attack surface cost less than the capability it provides?" For server workloads running functions and containers, virtio-net, virtio-block, virtio-vsock, virtio-rng, a serial console, and a reboot signal are sufficient. The catalog is closed precisely because that boundary is defensible.

Sources And Further Reading

- Firecracker source tree (`main`, June 2026): https://github.com/firecracker-microvm/firecracker
- `FAQ.md` — device inventory and design philosophy: https://github.com/firecracker-microvm/firecracker/blob/main/FAQ.md
- `docs/design.md` — threat model and i8042 purpose: https://github.com/firecracker-microvm/firecracker/blob/main/docs/design.md
- `docs/entropy.md` — virtio-rng design and `aws-lc-rs` rationale: https://github.com/firecracker-microvm/firecracker/blob/main/docs/entropy.md
- `docs/vsock.md` — vsock backend, UDS routing, connection protocol: https://github.com/firecracker-microvm/firecracker/blob/main/docs/vsock.md
- `docs/mmds/mmds-design.md` — Dumbo, V1/V2 authentication design: https://github.com/firecracker-microvm/firecracker/blob/main/docs/mmds/mmds-design.md
- `docs/mmds/mmds-user-guide.md` — MMDS operator reference: https://github.com/firecracker-microvm/firecracker/blob/main/docs/mmds/mmds-user-guide.md
- `docs/kernel-policy.md` — i8042 boot parameter policy: https://github.com/firecracker-microvm/firecracker/blob/main/docs/kernel-policy.md
- `CHANGELOG.md` — version history for rate limiter, virtio-rng, v1.16.0 changes: https://github.com/firecracker-microvm/firecracker/blob/main/CHANGELOG.md
- virtio-MMIO transport source — `src/vmm/src/devices/virtio/transport/mmio.rs`: https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/devices/virtio/transport/mmio.rs
- MMIO device manager — `src/vmm/src/device_manager/mmio.rs`: https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/device_manager/mmio.rs
- Rate limiter source — `src/vmm/src/rate_limiter/mod.rs`: https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/rate_limiter/mod.rs
- Rate limiter persist — `src/vmm/src/rate_limiter/persist.rs`: https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/rate_limiter/persist.rs
- OpenAPI spec — `src/firecracker/swagger/firecracker.yaml`: https://github.com/firecracker-microvm/firecracker/blob/main/src/firecracker/swagger/firecracker.yaml

- OASIS virtio 1.2 CS01 — transport register map (§4.2), initialization sequence (§3.1.1), RNG device (§5.4): https://docs.oasis-open.org/virtio/virtio/v1.2/cs01/virtio-v1.2-cs01.html
- Issue #650 — vsock vhost exclusion rationale: https://github.com/firecracker-microvm/firecracker/issues/650
- Issue #1268 — legacy GSI IRQ limit and sharing: https://github.com/firecracker-microvm/firecracker/issues/1268
- Issue #259 — rate limiter OverConsumption deadlock fix: https://github.com/firecracker-microvm/firecracker/issues/259
- PR #5789 — virtio-pmem rate limiting (v1.16.0): https://github.com/firecracker-microvm/firecracker/pull/5789
- PR #5824 — serial console rate limiting (v1.16.0): https://github.com/firecracker-microvm/firecracker/pull/5824
- AWS Open Source Blog — Announcing Firecracker (2018): https://aws.amazon.com/blogs/opensource/firecracker-open-source-secure-fast-microvm-serverless/

Chapter 15: Boot And Configuration

By the time you call `PUT /actions` with `action_type: "InstanceStart"`, the `firecracker` process has been running for perhaps a dozen milliseconds and has done almost nothing. It opened `/dev/kvm`, issued `KVM_CREATE_VM`, and started listening on a Unix domain socket. The guest CPU has never executed an instruction. What happens between process start and that first instruction is the subject of this chapter: a sequence of HTTP/1.1 API calls over a Unix socket that assembles every parameter the VMM needs to build a virtual machine from scratch — and hands control to the guest kernel in under 125 ms.

That number, 125 ms from `InstanceStart` to the start of the guest user-space `/sbin/init` process, is not an aspiration. It is a contractual requirement stated in `SPECIFICATION.md`, measured under specific conditions: serial console disabled, minimal kernel, minimal root filesystem. Clearing it constrains every API design decision. Configuration has to be complete before boot, not discovered during it. The API has to be stateless across calls, not transactional. The device model has to start without probing for hardware that does not exist.

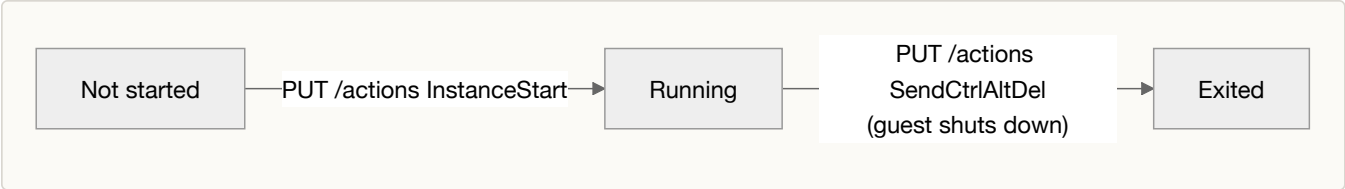
The Transport Layer

Firecracker does not expose its API on a TCP port. Every request travels over a **Unix domain socket (UDS)** using HTTP/1.1, with `Content-Type: application/json` on both sides. The socket path is operator-chosen and passed to the `firecracker` binary via `--api-sock`; there is no default. The OpenAPI 2.0 swagger document at `src/firecracker/swagger/firecracker.yaml` declares `host: localhost` and `schemes: [http]` as artifacts of the spec format, not as TCP configuration.

The choice of a Unix socket over a TCP port is not accidental. The jailer creates the socket inside the chroot before `exec`ing `firecracker`; the host process that manages the VM holds the socket path and is the only caller that can reach it. There is no exposed port for a network scanner to find, no bind address to misconfigure, and no TCP stack overhead on what is already a localhost path. The swagger spec version in the repository as of June 2026 is `1.17.0-dev`.

The Pre-Boot State Machine

The API endpoints divide cleanly into two classes based on when they are valid. Most configuration calls are **pre-boot only**: `PUT /boot-source`, `PUT /drives/{id}`, `PUT /machine-config`, and `PUT /network-interfaces/{id}` all refuse to execute after the VM has started. `PUT /actions` with `action_type: "InstanceStart"` is the transition that moves the VMM from the `"Not started"` state to `"Running"`. After that crossing, a different set of endpoints becomes valid: `PATCH /drives/{id}`, `PATCH /network-interfaces/{id}`, and `PUT /actions` with `action_type: "FlushMetrics"` or `"SendCtrlAltDel"`.



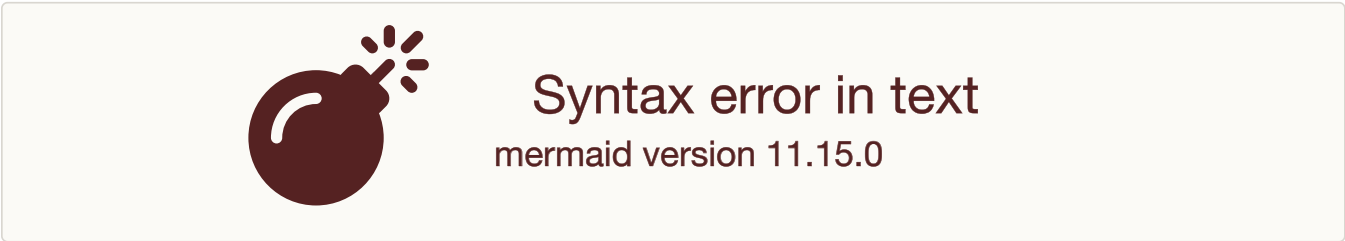
`GET /` is valid in all states. It returns an `InstanceInfo` object with four required fields: `app_name` (always the string `"Firecracker"`), `id` (the operator-assigned or auto-generated instance identifier), `state` (one of `"Not started"`, `"Running"`, or `"Paused"`), and `vmm_version` (the build version of the running `firecracker` binary). The state strings matter when you are scripting: `"Not started"` is two words with a space, and the field for the build version is `vmm_version`, not `firecracker_version`.

Configuring The Boot Source

`PUT /boot-source` sets three fields. Only one is required.

`kernel_image_path` is an absolute path on the host filesystem to the uncompressed kernel image. On x86_64 that means `vmlinux`, an ELF binary typically in the 4–8 MB range. On aarch64 it means a PE-formatted `Image` file. The kernel must be uncompressed because Firecracker does not implement a bootloader — it loads the kernel image directly into guest memory and launches it via one of two boot protocols. When the kernel exports a PVH entry point (`pvh_boot_cap`), Firecracker uses PVH boot (`BootProtocol::PvhBoot`), which is the protocol its tuned kernels use. If no PVH entry point is present, Firecracker falls back to the Linux 64-bit boot protocol (`BootProtocol::LinuxBoot`). The guest memory load region starts at `0x100000` (`HIMEM_START`), but the entry address differs between the two protocols. There is no GRUB, no BIOS, no real-mode stage.

`boot_args` is the kernel command line. The API does not enforce a default; if you omit the field, no command-line arguments are passed. In practice, a working command line for a block-device boot looks like:



The `pci=off` suppresses PCI bus enumeration entirely — there is no PCI bus to find, but without that flag some kernel configurations will wait for one. The `reboot=k` tells the kernel to send a keyboard controller reset on reboot, which Firecracker intercepts and converts to a clean VMM exit. `nomodules` is appropriate here because a Firecracker guest kernel carries no loadable modules; the compiled-in device drivers are all that exist. `firectl`, the thin Go wrapper around the Firecracker API, uses these as its default when `--kernel-opts` is not specified.

`initrd_path` is optional. Set it to a host path to load an initramfs image into guest memory before jumping to the kernel entry point. Set it to `null` or omit it entirely for a direct boot into the root block device. The distinction matters for the required kernel configuration: initrd-only boot does not need `CONFIG_ACPI=y`, `CONFIG_PCI=y`, or `CONFIG_VIRTIO_BLK=y`, while a block-device boot requires all three.

`PUT /boot-source` accepts only `PUT` — no `PATCH`, no `GET`. A second `PUT` before `InstanceStart` replaces the previous configuration entirely. The semantics are idempotent in the replace-the-whole-thing sense, not in the merge-fields sense.

Machine Configuration

`PUT /machine-config` — or its field-merging sibling `PATCH /machine-config`, which is also pre-boot only — controls the virtual hardware the guest sees before it executes its first instruction: how many CPUs, how much memory, whether hyperthreading is advertised, and which CPU template shapes the CPUID leaves.

`vcpu_count` is required. The minimum is 1; the maximum is 32, a hard limit enforced in the Firecracker source and declared with `maximum: 32` in the swagger schema. The `KVM_CREATE_VCPU` ioctl is called once per vCPU, and each vCPU becomes one host thread — 32 is therefore also the thread-count ceiling on the VMM process.

`mem_size_mib` is required. The swagger schema carries no `minimum` constraint. The practical floor is 128 MiB, the value the FAQ documents as its default; below that, the Linux boot sequence may fail before reaching `init`. The effective ceiling is set by `KVM_MEM_MAX_NR_PAGES`, a KVM constant that depends on host kernel version and architecture.

`smt` is a boolean, optional, defaulting to `false`, and valid only on x86_64. Setting it to `true` on aarch64 returns a 400. When `true`, Firecracker presents a CPU/cache topology to the guest that reflects simultaneous multithreading — concretely, the topology structures visible via `cpuid` leaf `0xb` will indicate that threads share a core. This field was named `ht_enabled` before Firecracker v1.0.0; the rename to `smt` happened in that release, and the field became optional with a default of `false`.

> ***Before setting `smt: true` in production:*** *the Firecracker production host setup guide recommends disabling SMT at the host level in multi-tenant deployments because "SMT is frequently a precondition for speculation issues utilized in side channel attacks such as Spectre variants and MDS." The host kernel exposes SMT control at `/sys/devices/system/cpu/smt/control`; writing `off` to that file disables SMT host-wide. The `forceoff` and `notsupported` values are read-only and set by the kernel, not the operator. Guest-visible SMT and host SMT are independent settings; you can present SMT to the guest while running the host with SMT disabled, but the presented topology will not reflect real hardware.*

`track_dirty_pages` enables KVM's dirty bitmap for the guest's memory slot. When `true`, `KVM_GET_DIRTY_LOG` can be used to identify which 4 KiB pages the guest has written since the last flush — the mechanism that underpins live migration and snapshots. Enabling dirty tracking at 4 KiB granularity has a measurable cost when combined with huge pages, discussed below.

`huge_pages` takes the string enum `"None"` (the default) or `"2M"`. When set to `"2M"`, Firecracker backs the guest memory slot with 2 MB hugetlbfs pages instead of 4 KiB base pages. 1 GB pages are not supported. This feature was added as a developer preview in Firecracker v1.7.0. The boot time improvement in Firecracker's own benchmarks reached up to 50%, because 2 MB pages mean fewer TLB misses during the kernel's initial page walks across guest memory.

Three caveats apply to `huge_pages: "2M"`. First, the host must pre-allocate a large enough 2 MB huge page pool before the VM starts; Firecracker maps guest memory with `MAP_NORESERVE`, so an undersized pool causes `SIGBUS` at access time, not at map time. Second, snapshots of a hugepage-backed VM can only be restored via userfaultfd (UFFD); the standard `mmap`-based restore path does not support them. Third — and this interaction is easy to miss — setting `track_dirty_pages: true` alongside `huge_pages: "2M"` destroys the performance benefit: KVM reverts to 4 KiB PTE granularity for dirty tracking, which is exactly what 2 MB pages were bought to avoid.

CPU Templates

`cpu_template` accepts a string enum selecting a **static CPU template**, which shapes the CPUID leaves and MSRs the guest sees. Static templates have been deprecated since Firecracker v1.5.0 in favor of `PUT /cpu-config`, which accepts a JSON structure with architecture-specific overrides. If both a static template and a custom template are configured, whichever was set most recently wins.

The official documentation is explicit that "CPU templates shall not be used as a security protection against malicious guests." They exist for fleet homogeneity, not for isolation.

The available static templates and their target hardware are:

Template	Architecture	Target host CPUs
`C3`	x86_64 (Intel)	Skylake, Cascade Lake, Ice Lake
`T2`	x86_64 (Intel)	Skylake, Cascade Lake, Ice Lake
`T2S`	x86_64 (Intel)	Skylake, Cascade Lake
`T2CL`	x86_64 (Intel)	Cascade Lake, Ice Lake
`T2A`	x86_64 (AMD)	Milan
`V1N1`	aarch64 (ARM)	Neoverse V1, presented as Neoverse N1
`None`	both	No template applied

`T2CL` and `T2A` together present a homogeneous instruction-set surface across Intel Cascade Lake and AMD Milan hosts in a mixed fleet — the same guest binary runs identically on either without branching on CPU vendor.

On x86_64, what these templates actually do is set specific CPUID leaf values and MSR contents. The `C3` template sets CPUID leaf `0x1 EAX` to `0x000306e4`, which decodes to family 6, model `0x3e` — the Ivy Bridge / Xeon E5 v2 signature. `T2` sets the same leaf to `0x000306F2` (family 6, model `0x3f`, Haswell). `T2S` sets CPUID leaf `0x1 EAX` to the same value — `0x000306F2`, family 6, model `0x3f` — so T2 and T2S present an identical CPU version identifier to the guest. The distinction between them is in ECX/EDX feature bits on other leaves. `T2S` additionally writes MSR `IA32_ARCH_CAPABILITIES` at address `0x10A` to `0x000000000C080C4C`, exposing the specific capability bits needed for safe snapshot migration between Skylake and Cascade Lake hosts.

The `V1N1` aarch64 template modifies four system registers to make a Neoverse V1 host look like a Neoverse N1 to the guest: `ID_AA64PFR0_EL1` (clears SVE at bits 35:32 and DIT at bits 51:48), `ID_AA64ISAR0_EL1` (clears SHA3, SM3, SM4, ASIMDFHM, FLAGM, and RND; sets SHA2 to `0b0001`), `ID_AA64ISAR1_EL1` (clears JSCVT, FCMA, BF16, DGH, and I8MM; sets DPB to `0b0001` and LRCPC to `0b0001`), and `ID_AA64MMFR2_EL1` (clears USCAT at bits 35:28). The effect is that code checking these ID registers on a V1 host sees the N1 feature set — the guest binaries compiled for N1 run without modification.

Regardless of which template is selected, Firecracker applies CPUID normalization on x86_64 **after** template application, so a template-set bit can be overwritten by normalization. Leaves touched on all CPUs include `0x0`, `0x1`, `0xb`, `0x80000005`, and `0x80000006`. On Intel hosts, normalization additionally touches leaves `0x4`, `0x6`, `0x7`, `0xa`, `0x1f`, and `0x80000002` – `0x80000004`. On AMD, it touches `0x7`, `0x80000001`, `0x80000002` – `0x80000004`, `0x80000008`, `0x8000001d`, and `0x8000001e`. The practical effects include disabling Intel Turbo Boost, disabling performance monitoring counters, setting the `HYPERVISOR` bit in leaf `0x1 ECX`, and enabling `TSC_DEADLINE` for APIC timer use.

For operators who need finer control than the static templates offer, `PUT /cpu-config` accepts a JSON body with `cpuid_modifiers`, `msr_modifiers`, and `kvm_capabilities` on x86_64, or `reg_modifiers`, `vcpu_features`, and `kvm_capabilities` on aarch64. Individual bits are expressed with a bitmap notation where `"x"` means pass-through (inherit from hardware), `"0"` forces the bit clear, and `"1"` forces it set. Underscore characters in bit strings are visual separators only.

Drives And The Root Filesystem

Every storage device the guest sees is a **virtio-block** device. Firecracker implements the OASIS virtio 1.2 specification's block device type (Device ID 2, section 5.2). The guest addresses all block devices at sector granularity of 512 bytes; the `sector` field in every virtio-blk request is a 512-byte offset into the device.

`PUT /drives/{drive_id}` attaches a drive. The `drive_id` path parameter and the matching body field together identify the device. `is_root_device: true` marks the drive that the guest kernel will find as `/dev/vda` — the paravirtualized block device that holds the root filesystem. Exactly one drive may carry `is_root_device: true`.

`path_on_host` is the absolute host filesystem path to the backing block file — a raw image, an ext4 file, a squashfs image. The file must exist and be readable by the `firecracker` process at the time of the `PUT` call.

`is_read_only` sets the virtio feature bit `VIRTIO_BLK_F_RO` (bit 5). When this bit is advertised, write requests from the guest driver must fail with `VIRTIO_BLK_S_IOERR`; the host-side device emulator refuses to pass any write through to the backing file. The guest kernel detects the read-only state during feature negotiation, before any I/O is attempted.

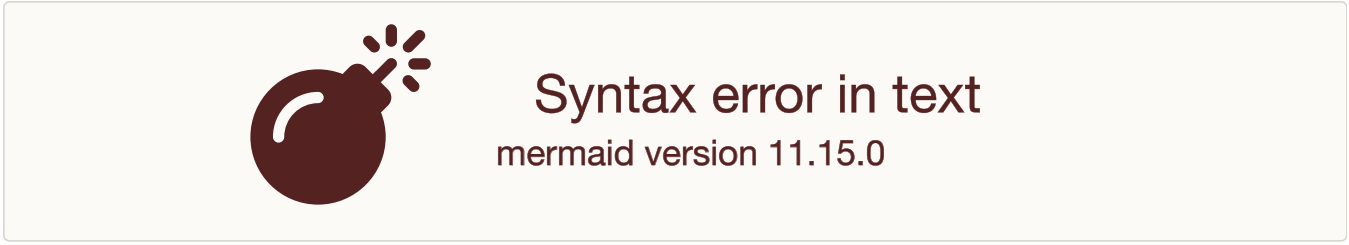
`io_engine` selects between `"Sync"` (the default, using standard blocking file I/O) and `"Async"` (added in Firecracker v1.0.0, backed by `io_uring`). The async engine reduces per-operation latency under load by avoiding the host thread context switch per I/O, at the cost of requiring `io_uring` support in the host kernel (Linux 5.1 or later).

`cache_type` selects `"Unsafe"` (default, writeback not guaranteed to be flushed) or `"Writeback"`, which maps to the `VIRTIO_BLK_F_CONFIG_WCE` feature bit (bit 11) and enables explicit writeback cache semantics. The naming is deliberate: `"Unsafe"` is the correct choice when the backing file is on a fast local SSD whose durability you do not need to guarantee; `"Writeback"` is correct when the guest's writes must survive a host crash.

`PATCH /drives/{drive_id}` is **post-boot only**. It allows updating `path_on_host` and the rate limiter fields, and nothing else. `is_read_only` and `is_root_device` cannot be patched. This is not hot-plug of a new device; the virtio device the guest sees is the same device, with a different backing file on the host side. The use case is swapping the data volume of a running VM without the guest noticing a device disappear and reappear.

The Root Filesystem As A Block Device

The canonical setup for a Firecracker VM assigns an ext4 image to the root drive and passes appropriate boot arguments to the kernel:



```
PUT /boot-source
{
  "kernel_image_path": "/path/to/vmlinux",
  "boot_args": "root=/dev/vda console=ttyS0 reboot=k panic=1 pci=off"
}
```

The `root=/dev/vda` argument tells the kernel which device holds the root filesystem. Because virtio-blk devices appear as `/dev/vda`, `/dev/vdb`, and so on in the order they were attached, the root drive is always `/dev/vda` when `is_root_device: true` was set.

Read-Only Base Plus Overlay

When thousands of VMs share the same base operating system image, copying a multi-gigabyte root filesystem into each VM's backing file is expensive in both time and disk space. The standard pattern is to use a single read-only base image, attach a per-VM overlay drive for writes, and wire them together inside the guest with the Linux overlay filesystem.

The setup uses two drives:

```
flowchart TB
  A["base-rootfs.squashfs\n(is_read_only: true)\nguest: /dev/vda"] --> C["overlay
filesystem in guest\n(mount -t overlay)"]
  B["overlay-vm-N.ext4\n(is_read_only: false)\nguest: /dev/vdb"] --> C
  C --> D["pivot_root → new /"]
```

1. Attach the base image — typically a squashfs image — as the root drive with `is_read_only: true`. The guest will see it as `/dev/vda`.
2. Attach a per-VM sparse ext4 file as a second drive with `is_read_only: false`. The guest sees it as `/dev/vdb`.
3. Set `boot_args` to include `init=/sbin/overlay-init overlay_root=vdb`.
4. The `/sbin/overlay-init` script in the base image mounts the overlay ext4 at `/overlay`, then calls `mount -t overlay overlay -o lowerdir=/,upperdir=/overlay/root,workdir=/overlay/work /mnt`, and calls `pivot_root` to switch to the merged view. After `pivot_root`, the running system's `/` is the overlay, writes go to the ext4 file on `/dev/vdb`, and the squashfs base is never modified.

The directories `/overlay/root`, `/overlay/work`, `/mnt`, and `/rom` must be pre-created inside the squashfs image, because the base filesystem is read-only at runtime and cannot be written during init.

For a **temporary** overlay — state lost on VM termination — replace `overlay_root=vdb` with `overlay_root=ram` or omit it; a tmpfs layer handles writes and vanishes with the process. For a **persistent** overlay, `overlay_root=vdb` directs writes to the ext4 sparse file, which survives VM termination. The sparse file starts near empty and grows only as the guest writes; a typical minimal workload accumulates tens of megabytes per VM rather than gigabytes.

Network Interfaces

`PUT /network-interfaces/{iface_id}` connects the guest to the host network. Firecracker requires the TAP device to already exist on the host before the API call; the VMM does not create it. The `host_dev_name` field names the TAP interface, and Firecracker opens it by name and holds the file descriptor for the lifetime of the VM.

> ***Before running this step:*** *creating a TAP device and attaching it to a bridge requires root (or `CAP_NET_ADMIN`) on the host. The sequence is: `ip tuntap add dev tap0 mode tap` and `ip link set tap0 up`. When running under the jailer, the jailer creates the TAP before exec'ing `firecracker`, because the chroot eliminates access to `/dev/net/tun` after exec.*

`guest_mac` is optional. If omitted, Firecracker generates a MAC address deterministically from the interface index. Specifying it explicitly is necessary when the address must be stable across VM recreation or must match a DHCP reservation.

`mtu` was added in Firecracker v1.16.0. It sets the MTU advertised to the guest via `VIRTIO_NET_F_MTU`. The valid range is 68–65535. If omitted, the guest uses the default MTU negotiated by the virtio-net driver. In practice, setting this to match the host's physical interface MTU (typically 1500, or 9000 for jumbo frames) avoids silent fragmentation at the host TAP boundary.

`rx_rate_limiter` and `tx_rate_limiter` attach token-bucket rate limiters to the ingress and egress paths. Both fields accept a `RateLimiter` object containing two independent `TokenBucket` sub-objects: `bandwidth` (measured in bytes per second) and `ops` (measured in operations per second). Each bucket has three fields: `size` sets the bucket capacity, `one_time_burst` sets an initial fill that allows bursting above the steady-state rate, and `refill_time` sets the time in milliseconds to refill the bucket from zero to `size`. The steady-state rate is `size / refill_time`. Setting both `size` and `refill_time` to `0` disables the limiter.

`PATCH /network-interfaces/{iface_id}` is **post-boot only** and accepts only the rate limiter fields. The `host_dev_name` and `guest_mac` cannot be changed on a running VM; those are part of the virtio device identity that the guest negotiated at boot.

Logging

Firecracker's logger does not emit a word until it is explicitly initialized. The process starts, opens its API socket, and waits — silently. `PUT /logger` performs that initialization, and it can only be called once. A second call before the logger is active returns 400; once initialized, there is no API to reconfigure it.

> ***Before calling `PUT /logger`:*** *the destination file or named pipe must already exist on the filesystem. For a named pipe: `mkfifo /tmp/fc-log.fifo`. For a regular file: `touch /tmp/fc-log.txt`. The `firecracker` process must be able to open the path for writing; inside the jailer's chroot, the path must be inside the chroot directory.*

The logger schema carries five fields. `log_path` is the only one without a default; it must name a pre-existing file or named pipe. `level` defaults to `"Info"` and accepts `Error`, `Warning`, `Info`, `Debug`, `Trace`, and `Off` (case-insensitive). `show_level` (default `false`) prepends the level string to each log line. `show_log_origin` (default `false`) prepends the Rust source file path and line number — useful when debugging, expensive when processing logs at scale. `module` filters log output to a specific Rust module path, for example `"api_server::request"` to trace only incoming API traffic.

The logger can also be configured via CLI flags — `--log-path`, `--level`, `--show-level`, `--show-log-origin` — passed to the `firecracker` binary at startup. The CLI path and the `PUT /logger` API path are mutually exclusive; use one or the other per process.

Firecracker produces no log output between `jailer` startup and logging-system initialization — a window that covers seccomp filter installation, the chroot, and the initial API socket setup. If the named pipe fills because the consumer is slow, Firecracker drops log entries silently and increments the `lost-logs` metric counter. A blocked log consumer starves you of operational visibility without any visible error.

Metrics

Where the logger emits human-readable lines, the metrics subsystem emits machine-readable JSON. `PUT /metrics` initializes it, and like the logger, it can only be called once — a second call returns 400.

> ***Before calling `PUT /metrics`:*** *the destination must already exist:* `mkfifo /tmp/fc-metrics.fifo` *or* `touch /tmp/fc-metrics.txt`.

The schema has one field: `metrics_path`, the path to the named pipe or file that receives JSON metric objects.

Metrics are flushed in two ways. Automatically, every 60 seconds, Firecracker writes one JSON object containing all current counters. On demand, `PUT /actions` with `action_type: "FlushMetrics"` triggers an immediate flush without waiting for the 60-second timer.

Each flush emits a single JSON object with 21 top-level category keys: `api_server`, `balloon`, `block`, `deprecated_api`, `entropy`, `get_api_requests`, `i8042`, `latencies_us`, `logger`, `mmds`, `net`, `patch_api_requests`, `put_api_requests`, `rtc`, `seccomp`, `signals`, `uart`, `vcpu`, `vhost_user_block`, `vmm`, and `vsock`. All 21 categories are emitted on every flush regardless of whether the associated device is present; if you have no balloon device, the `balloon` category still appears, with zero values. The naming convention within each category is consistent: counters with `_bytes` or `_bytes_count` are byte quantities, `_ms` is milliseconds, `_us` is microseconds, and bare names without a suffix are event counts.

The metrics configuration is not captured in a VM snapshot and is not restored from one. A VM restored from a snapshot must reconfigure its metrics endpoint from scratch.

The back-pressure behavior mirrors the logger: if the named pipe is full, metric flush events are dropped and the `lost-metrics` counter is incremented.

Starting The VM

With boot source, drives, network interfaces, and optionally logger and metrics configured, the VM is ready to start. The call is:

```
PUT /actions
{
  "action_type": "InstanceStart"
}
```

This is the point of no return. `InstanceStart` can only be called once per `firecracker` process lifecycle; there is no API to stop and restart a running VM. The VMM issues `KVM_SET_USER_MEMORY_REGION` to map the guest memory into the VM, issues `KVM_CREATE_VCPU` for each configured vCPU, loads the kernel into guest memory starting at `0x100000` (`HIMEM_START`), selects the boot protocol (PVH when the kernel exports a PVH entry point, Linux 64-bit otherwise), sets up the initial register state for that protocol, and calls `KVM_RUN`. The guest CPU begins executing.

From that moment, the clock is running against the 125 ms SLA. Under the specified conditions — serial console absent from `boot_args`, Firecracker-tuned kernel, minimal root filesystem — the kernel traverses its init path, probes the virtio-MMIO devices (no PCI scan, no timeout), mounts the root filesystem, and forks `/sbin/init` inside that budget.

`PUT /actions` also exposes two post-boot operations. `"FlushMetrics"` triggers an immediate metrics flush as described above. `"SendCtrlAltDel"` emits a CTRL+ALT+DEL sequence through the emulated i8042 keyboard controller; this is x86_64 only and requires the guest kernel to have been compiled with `CONFIG_SERIO_I8042` and `CONFIG_KEYBOARD_ATKBD`. The guest receives the signal, initiates its orderly shutdown, and when the guest CPU issues a reset instruction, Firecracker exits the process. There is no API-level pause or resume of a running VM without snapshot machinery (covered in Chapter 16).

The Config-File Shortcut

Issuing API calls individually requires a caller that can speak HTTP/1.1 over a Unix socket and sequence its requests correctly. For scripting and for integration tests, `firecracker` accepts a `--config-file` flag pointing to a JSON file that expresses the same configuration in one shot.

The file's JSON structure mirrors the API endpoints, with one naming convention worth internalizing: **top-level keys use hyphens** (`boot-source`, `machine-config`, `network-interfaces`, `cpu-config`) while **nested field names use snake_case** (`kernel_image_path`, `vcpu_count`, `host_dev_name`). The hyphens come from API URL path segments; the underscores come from JSON body field names. Mixing these up produces a parse error with no helpful message.

At minimum, the file must contain `boot-source.kernel_image_path` and at least one entry in `drives` with `is_root_device: true`. All other sections are optional. When `--config-file` is supplied without `--no-api`, the Unix domain socket is still created and remains available for subsequent API calls — so config-file and runtime API are not mutually exclusive.

The full set of top-level keys the config file accepts as of Firecracker v1.14 and later: `boot-source`, `drives`, `machine-config`, `cpu-config`, `balloon`, `network-interfaces`, `vsock`, `logger`, `metrics`, `mmds-config`, `entropy`, `pmem`, and `memory-hotplug`.

```
sequenceDiagram
    participant C as Caller
    participant F as "firecracker"
    participant K as KVM

    C->>F: --api-sock /run/fc.sock [--config-file]
    Note over F: KVM_CREATE_VM, API socket ready (~12 ms)
    C->>F: PUT /logger
    C->>F: PUT /metrics
    C->>F: PUT /machine-config
    C->>F: PUT /boot-source
    C->>F: PUT /drives/rootfs
    C->>F: PUT /network-interfaces/eth0
    C->>F: PUT /actions {"action_type":"InstanceStart"}
    F->>K: KVM_SET_USER_MEMORY_REGION
    F->>K: KVM_CREATE_VCPU (×vcpu_count)
    F->>K: KVM_RUN
    Note over F: Guest executing, state = Running
```

Beyond The Core Four

Several additional pre-boot endpoints extend the VM beyond a bare compute node: host-guest communication without a network stack, memory pressure signaling, persistent memory, dynamic memory pools, and the metadata service that replaces IMDS in the guest.

`PUT /vsock` attaches a virtio-vsock device for host-guest communication without a network stack. `guest_cid` (minimum 3; CIDs 0–2 are reserved) and `uds_path` are required. The vsock device appears to the guest as a standard AF_VSOCK socket endpoint.

`PUT /balloon` attaches the virtio-balloon device. The required fields are `amount_mib` and `deflate_on_oom`. The optional `stats_polling_interval_s` (0 to disable) controls how often the balloon driver reports guest memory statistics to the VMM. The balloon device cannot reclaim hugepage-backed guest memory: the 4 KiB granularity of the balloon protocol does not align with 2 MB huge pages.

`PUT /pmem/{id}`, added in Firecracker v1.14.0, attaches a virtio-pmem device exposing a host file as persistent memory in the guest. The guest kernel requires `CONFIG_VIRTIO_PMEM`, `CONFIG_LIBNVDIMM`, `CONFIG_BLK_DEV_PMEM`, and `CONFIG_DAX`. Devices appear as `/dev/pmem0`, `/dev/pmem1`, and so on. The

backing file must be 2 MB aligned.

`PUT /hotplug/memory`, also added in v1.14.0, allocates a pool of memory available for dynamic adjustment via `PATCH /hotplug/memory` while the VM is running. The required field is `total_size_mib`, which must be a multiple of `slot_size_mib` (default 128 MiB). The guest kernel must support `CONFIG_VIRTIO_MEM` (Linux 5.16 or later on x86_64, 5.18 or later on aarch64). Memory within this pool can be offered to the guest with `requested_size_mib` at runtime; the guest kernel decides how much to accept based on `memhp_default_state`. Boot memory and hotplug memory are separate; only memory allocated in the hotplug pool can be dynamically adjusted.

`PUT /mmds-config` configures the MicroVM Metadata Service — the in-process datastore the guest queries at `169.254.169.254` via HTTP. It requires `network_interfaces`, an array of `iface_id` strings naming which network interfaces MMDS listens on. The optional `version` field selects `"V1"` (default, no authentication) or `"V2"` (token-based sessions). MMDS is the subject of Chapter 17.

Sources And Further Reading

- Firecracker OpenAPI spec (v1.17.0-dev): https://raw.githubusercontent.com/firecracker-microvm/firecracker/main/src/firecracker/swagger/firecracker.yaml
- Firecracker specification (boot SLA, pre-boot state machine): https://github.com/firecracker-microvm/firecracker/blob/main/SPECIFICATION.md
- Getting started guide: https://github.com/firecracker-microvm/firecracker/blob/main/docs/getting-started.md
- Reference VM config (`tests/framework/vm_config.json`): https://github.com/firecracker-microvm/firecracker/blob/main/tests/framework/vm_config.json
- CPU templates documentation: https://github.com/firecracker-microvm/firecracker/blob/main/docs/cpu_templates/cpu-templates.md
- CPUID normalization documentation: https://github.com/firecracker-microvm/firecracker/blob/main/docs/cpu_templates/cpuid-normalization.md
- Static template sources (x86_64 — `c3.rs`, `t2.rs`, `t2s.rs`, `t2cl.rs`, `t2a.rs`): https://github.com/firecracker-microvm/firecracker/tree/main/src/vmm/src/cpu_config/x86_64/static_cpu_templates
- Static template sources (aarch64 — `v1n1.rs`): https://github.com/firecracker-microvm/firecracker/tree/main/src/vmm/src/cpu_config/aarch64/static_cpu_templates
- Huge pages documentation: https://github.com/firecracker-microvm/firecracker/blob/main/docs/hugepages.md
- Production host setup guide: https://github.com/firecracker-microvm/firecracker/blob/main/docs/prod-host-setup.md
- Logger documentation: https://github.com/firecracker-microvm/firecracker/blob/main/docs/logger.md

- Metrics documentation: https://github.com/firecracker-microvm/firecracker/blob/main/docs/metrics.md
- Actions documentation: https://github.com/firecracker-microvm/firecracker/blob/main/docs/api_requests/actions.md
- PATCH block drive documentation: https://github.com/firecracker-microvm/firecracker/blob/main/docs/api_requests/patch-block.md
- PATCH network interface documentation: https://github.com/firecracker-microvm/firecracker/blob/main/docs/api_requests/patch-network-interface.md
- Firecracker CHANGELOG (v1.0.0 SMT rename, v1.5.0 static template deprecation, v1.7.0 huge pages, v1.14.0 pmem/hotplug, v1.16.0 MTU): https://github.com/firecracker-microvm/firecracker/blob/main/CHANGELOG.md
- Memory hotplug documentation: https://github.com/firecracker-microvm/firecracker/blob/main/docs/memory-hotplug.md
- virtio-pmem documentation: https://github.com/firecracker-microvm/firecracker/blob/main/docs/pmem.md
- Firecracker Discussion #3092 (vCPU count limit, memory limits): https://github.com/firecracker-microvm/firecracker/discussions/3092
- Firecracker Discussion #3061 (rootfs overlay pattern): https://github.com/firecracker-microvm/firecracker/discussions/3061
- firecracker-containerd root-filesystem documentation: https://github.com/firecracker-microvm/firecracker-containerd/blob/main/docs/root-filesystem.md
- firectl `options.go` (flag-to-API mapping): https://raw.githubusercontent.com/firecracker-microvm/firectl/main/options.go
- Linux SMT sysfs ABI (`/sys/devices/system/cpu/smt/control`): https://www.kernel.org/doc/Documentation/ABI/testing/sysfs-devices-system-cpu
- OASIS virtio 1.2 specification, section 5.2 (virtio-blk): https://docs.oasis-open.org/virtio/virtio/v1.2/csd01/virtio-v1.2-csd01.html

Chapter 16: Snapshot And Restore

Booting a microVM from scratch takes on the order of 200 milliseconds — the kernel decompresses, the init process brings up devices, the runtime initializes. For most batch workloads that cost is amortized. For an invocation-per-request serverless platform running millions of events per second, it is the difference between a product that works and one that does not.

The mechanism that eliminates cold-start latency is **snapshot and restore**: freeze a running microVM, write its complete state to disk, then resume copies of that state on demand. The restored microVM picks up exactly where the frozen one left off — the kernel is already running, the runtime is already warm, the JIT cache is already populated. Firecracker snapshot restore with file-backed memory achieves as little as 4 ms on an EC2 m5.12xlarge. AWS Lambda SnapStart used this mechanism to bring Java function cold starts from over six seconds to under 200 ms.

The mechanism is not free. When the same frozen state resumes into multiple clones, every clone inherits identical entropy pools, identical clock values, identical session tokens, and identical PRNG seeds. What looks like a performance optimization is also a security surface that has to be explicitly managed.

Two Files On Disk

Every `PUT /snapshot/create` request produces exactly two output files. Firecracker does not manage block device contents; that is the operator's responsibility — block device files must be flushed and synced by the guest before the VM is paused.

The **microVM state file** is named by the `snapshot_path` field. It holds everything that is not guest memory: the serialized `MicrovmState` struct covering device emulation state, KVM VM state, and per-vCPU register state. The **guest memory file** is named by `mem_file_path`. It is a flat file of exactly `mem_size_mib * 1024 * 1024` bytes: a complete or partial image of guest physical memory, depending on snapshot type.

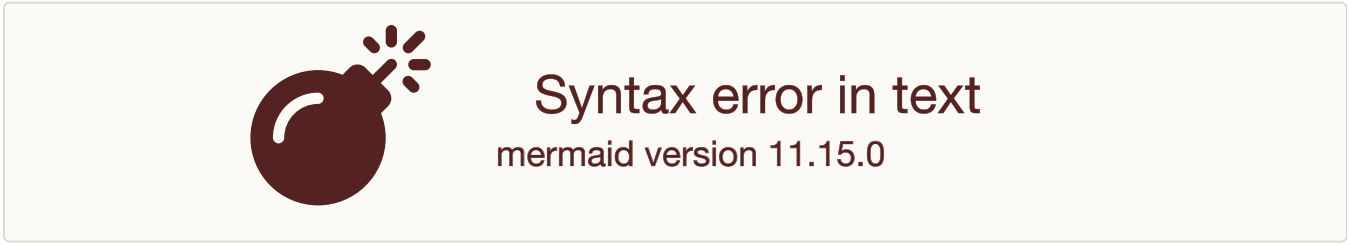
The API requires the VM to be paused before snapshotting:

```
PATCH /vm               {"state": "Paused"}
PUT /snapshot/create    {"snapshot_type": "Full", "snapshot_path": "/srv/snap/state",
"mem_file_path": "/srv/snap/mem"}
```

On the restore host, `PUT /snapshot/load` reads both files and leaves the microVM paused, ready to resume. If `resume_vm: true` is set in the load body, Firecracker issues the resume automatically; otherwise a separate `PATCH /vm {"state": "Resumed"}` is required.

The State File Format

The microVM state file has a fixed binary layout:

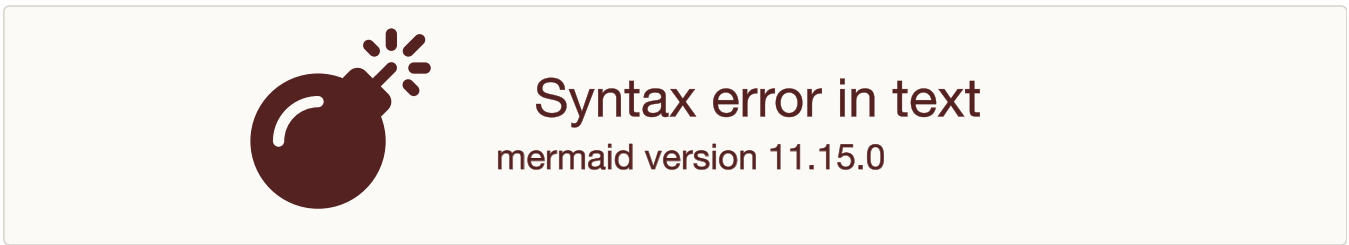


The magic ID is an architecture discriminant. On x86_64, `SNAPSHOT_MAGIC_ID = 0x0710_1984_8664_0000u64`; on aarch64, `0x0710_1984_AAAA_0000u64`. These constants live in `src/vmm/src/snapshot/mod.rs`. An attempt to load an aarch64 snapshot on an x86_64 host fails immediately at the magic check, before any deserialization occurs.

The version string encodes the snapshot format version in semver. As of `main` (June 2026), `SNAPSHOT_VERSION = Version::new(11, 0, 0)` in `src/vmm/src/persist.rs`. The compatibility rule is strict: the MAJOR version of the snapshot and the running Firecracker binary must match; the snapshot's MINOR must be less than or equal to the binary's MINOR; any PATCH version is accepted. Because Firecracker uses the `bitcode` crate rather than `bincode` for serialization — `bitcode` does not support backward-compatible schema evolution — every structural change to `MicrovmState` bumps the MAJOR version. Snapshot format versions are independent of Firecracker release versions; a v1.8.0 binary may write a format version 9.x snapshot.

The trailing eight bytes are a CRC-64 over all preceding bytes. The validation property is that `crc64(0, entire_file_bytes) == 0` when those bytes are the correct checksum. If the file is truncated, corrupted, or written by a different architecture, the check fails and restore is aborted. The deserializer enforces a 10 MB size limit (`SNAPSHOT_DESERIALIZATION_BYTES_LIMIT = 10_000_000`) to bound the cost of a malformed input.

The `bitcode` blob deserializes to a `MicrovmState`:



`VmInfo` carries the static configuration: `mem_size_mib`, SMT setting, CPU template, boot source, and huge page policy. `KvmState` records the KVM capability modifiers in effect when the snapshot was taken. `VmState` is architecture-specific: on x86_64 it includes IRQ routing tables, memory layout, and the kvmclock value. `DevicesState` covers the virtio block, net, and balloon devices; two device types are

explicitly excluded from the snapshot — the serial emulation state and the vsock backend. Neither can be meaningfully resumed; vsock listen sockets survive with updated CIDs, but open connections are torn down.

Full vs. Diff Snapshots

The `snapshot_type` field in `PUT /snapshot/create` selects between two memory-file strategies. Both produce sparse files; they differ in what pages are written.

Full Snapshots

A full snapshot calls `dump()` in `src/vmm/src/vstate/memory.rs`. It iterates all memory slots in order. Plugged (active) slots are written sequentially via `write_all_volatile`. Unplugged slots — memory that has been balloon-removed from the guest — are skipped with `SeekFrom::Current(slot_len_bytes)`, leaving a hole in the file at that range. The file is sparse: the kernel's page cache never populates those hole ranges, so the on-disk size reflects only the guest's active memory.

Diff Snapshots

A diff snapshot calls `dump_dirty()`. Instead of writing every plugged page, it unions two dirty-page sources and writes only the result.

The first source is the KVM dirty bitmap: `KVM_GET_DIRTY_LOG` returns a `u64` array for each memory slot, where bit `j` of element `i` represents page `(i * 64 + j)`. One ioctl call per slot. Clean pages are skipped with a seek forward; dirty pages are written at their correct file offset. The second source is Firecracker's internal `AtomicBitmap` tracking writes to virtio queue memory. KVM does not observe virtio queue activity directly at runtime — the device emulation layer in Firecracker does — so without the `AtomicBitmap`, virtio ring updates would silently go unrecorded.

After the diff file is written, Firecracker resets both sources: `reset_dirty_bitmap()` calls `get_dirty_log()` for each slot (clearing KVM's internal bitmap, not just reading it) and zeros the `AtomicBitmap`. It then calls `mark_virtio_queue_memory_dirty()` to pre-mark virtio queue memory dirty for the next diff cycle, since those pages will be written again before the next snapshot.

If `track_dirty_pages` was `false` at load time, KVM dirty-page logging was never enabled, and the diff path falls back to `mincore(2)` — the set of resident pages in memory — as its proxy for dirtiness. This fallback requires swap to be disabled; any swapped-out page would be misreported as clean.

When the output `mem_file_path` already exists and its size exactly matches `mem_size_mib * 1024 * 1024`, the diff patches only the dirty pages into that existing file. A size mismatch truncates and rewrites from scratch. An application can therefore take a diff snapshot after each phase of work and always merge into the same path without growing file count with depth.

```
flowchart LR
    A["PUT /snapshot/create\ntype=Diff"] --> B["dump_dirty()"]
    B --> C["KVM_GET_DIRTY_LOG\n(per memory slot)"]
    B --> D["AtomicBitmap\n(virtio queue writes)"]
    C --> E["union dirty set"]
    D --> E
    E --> F["write dirty pages\n(skip clean pages\nwith seek forward)"]
    F --> G["sparse mem_file_path"]
    G --> H["reset_dirty_bitmap()\nmark_virtio_queue_memory_dirty()"]
```

---

vCPU State: The Ioctl Sequence

Saving and restoring a vCPU is not a single ioctl. It is an ordered sequence where each step must precede the next because of field-level dependencies within KVM's internal state. The save and restore sequences in `src/vmm/src/arch/x86_64/vcpu.rs` make the ordering explicit.

Saving starts with `KVM_GET_MP_STATE` because that call triggers `kvm_apic_accept_events()` internally, which can modify the LAPIC state. If you read the LAPIC first, you may capture a stale value. Then come the general-purpose registers (`KVM_GET_REGS`), segment registers and control registers (`KVM_GET_SREGS`), extended processor state (`KVM_GET_XSAVE`), extended control registers (`KVM_GET_XCRS`), debug registers (`KVM_GET_DEBUGREGS`), and then the LAPIC (`KVM_GET_LAPIC` , `kvm_lapic_state` , 0x400 bytes). After the LAPIC: the TSC frequency via `get_tsc_khz()` , the CPUID configuration, and the MSRs in chunks via `KVM_GET_MSRS` . The sequence ends with `KVM_GET_VCPU_EVENTS` — pending exceptions, interrupts, NMIs, and SMIs — because it is affected by all of the preceding GETs and must come last to be accurate.

Restoring reverses the dependency direction. `KVM_SET_CPUID2` comes first because the BSP identity inside `set_mp_state` depends on CPUID. Then `KVM_SET_MP_STATE` , then `KVM_SET_REGS` — which clears pending exceptions and therefore must precede `KVM_SET_VCPU_EVENTS` . `KVM_SET_SREGS` restores the APIC base MSR and must precede `KVM_SET_LAPIC` . After `KVM_SET_XSAVE` , `KVM_SET_XCRS` , and `KVM_SET_DEBUGREGS` , the LAPIC is set with `KVM_SET_LAPIC` — which must follow `KVM_SET_SREGS` (APIC base) and must precede `KVM_SET_MSRS` (TSC deadline MSR reads the LAPIC timer mode). The MSRs are set in chunks via `KVM_SET_MSRS` . The sequence ends with `KVM_SET_VCPU_EVENTS` , which requires all other vCPUs to be paused — a constraint that holds during restore because the microVM begins life in the paused state.

The Restore Flow

`restore_from_snapshot()` in `src/vmm/src/persist.rs` executes in a fixed order. It reads and deserializes `MicrovmState` , validating the magic ID, version compatibility, and CRC-64 in the process. It applies any `network_overrides` from the load request — these remap tap device names in the

deserialized net device state, which is how a clone can be given a different host tap interface than the one the base snapshot used. It applies any `vsock_override`. It then calls `snapshot_state_sanity_check()`: at least one memory region must exist, at least one DRAM region must exist, and each DRAM region must have exactly one plugged slot.

Before proceeding to device reconstruction, Firecracker issues a cross-vendor warning. On x86_64 it calls `get_vendor_id_from_host()` and compares the host's CPU vendor string to the one recorded in the snapshot; on aarch64 it compares manufacturer IDs. It warns, rather than aborts, because some cross-model restores work in practice — but cross-architecture restores do not, and that failure mode is caught earlier by the magic ID.

With the sanity checks passed, Firecracker maps guest memory from the memory file and calls `builder::build_microvm_from_snapshot()` to reconstruct all devices from `DevicesState`, restore the KVM VM state, and run the per-vCPU restore sequence described above.

```
sequenceDiagram
    participant Op as operator
    participant FC as firecracker
    participant KVM as KVM
    participant Mem as guest memory

    Op->>FC: PUT /snapshot/load
    FC->>FC: read + validate MicrovmState
    FC->>FC: apply network_overrides
    FC->>FC: sanity_check()
    FC->>Mem: mmap(MAP_PRIVATE) or register UFFD
    FC->>KVM: rebuild devices, KVM_SET_USER_MEMORY_REGION
    loop per vCPU
        FC->>KVM: KVM_SET_CPUID2
        FC->>KVM: KVM_SET_MP_STATE
        FC->>KVM: KVM_SET_REGS / KVM_SET_SREGS / ...
        FC->>KVM: KVM_SET_VCPU_EVENTS
    end
    FC-->>Op: microVM paused, ready
```

Copy-on-Write Memory Backing

Guest memory is not copied into the restored microVM's address space. It is mapped. The choice of mapping strategy is the `backend_type` field in the `mem_backend` object of `PUT /snapshot/load`.

File Backend

With `backend_type: "File"`, Firecracker creates a `MAP_PRIVATE` mapping of the guest memory file. Pages are loaded on demand: when a vCPU reads a guest physical address that has not been touched yet, the kernel page-faults in the corresponding page from the file's page cache. When that page is first

written, the kernel allocates a private anonymous copy — the standard POSIX copy-on-write behavior. The snapshot file is never modified.

This is what makes fast clones cheap at the memory level. One base snapshot file, mapped `MAP_PRIVATE` by any number of restored microVM processes: all clones share the physical pages of the file via the page cache until each clone writes to them, at which point only the written pages escape into private allocation. The clone cost is proportional to the working set, not the total guest memory size.

UFFD Backend

With `backend_type: "Uffd"`, Firecracker outsources page-fault handling to a separate process over a Unix domain socket. The flow is:

1. Firecracker creates an anonymous mmap sized to the guest memory description.
2. It registers that region with a `userfaultfd` file descriptor via `UFFDIO_REGISTER` with mode `UFFDIO_REGISTER_MODE_MISSING`.
3. It connects to the handler process listening on the Unix socket named by `backend_path`, then sends the UFFD fd via `SCM_RIGHTS` ancillary data together with `GuestRegionUffdMapping` structs describing each guest memory region: `base_host_virt_addr`, `size`, `offset`, and `page_size`.
4. The handler blocks on the UFFD fd reading `UFFD_EVENT_PAGEFAULT` events, and responds to each with an `UFFDIO_COPY` that copies the requested page from wherever the handler stores its backing data.

On kernels >= 6.1, the UFFD fd is created via the `/dev/userfaultfd` device and the `USERFAULTFD_IOC_NEW` ioctl. Firecracker v1.5.0 added this path as the preferred route, with a fallback to the `userfaultfd(2)` syscall on kernels that predate the device file.

The handler process can back pages from anywhere — a local file, a distributed in-memory cache, an S3 bucket — without any change to Firecracker. That pluggability is why AWS Lambda SnapStart uses the UFFD path. The Lambda snapshot handler divides the guest memory file into 512 KB chunks and serves them from a two-level hierarchy: an L1 worker-local cache at roughly 1 ms per chunk and an L2 availability-zone-wide distributed cache at single-digit milliseconds per chunk, with S3 as the ultimate fallback. This tiered fetch is invisible to the guest; from the microVM's perspective, the page just arrives when needed.

One operational hazard: if the UFFD handler process crashes while the microVM is running, Firecracker hangs indefinitely on the next page fault. There is no timeout and no fallback mechanism. The handler must be treated as a critical daemon, not a best-effort service. There is also a balloon interaction to handle: when the guest unplugs a memory range (balloon inflation), the UFFD subsystem emits `UFFD_EVENT_REMOVE` for that range. The handler must zero those pages; failing to do so leaves stale data accessible after the balloon is deflated again.

Fast Clones And Cold-Start Patterns

The snapshot-and-restore machinery was designed for cloning. The canonical pattern is:

Boot one microVM. Run the application through initialization: load the runtime, warm the JIT, prime the caches. Then pause the VM and take a full snapshot. That snapshot becomes the **base image** shared across all future instances. For each invocation, restore a new microVM from that base image using a `MAP_PRIVATE` file mapping. The clone's memory diverges from the base only as the invocation runs; unused pages are never faulted in.

At steady state, a fleet of clones shares the physical pages of a single base memory file through the page cache, with each clone contributing only the private anonymous pages it has dirtied. On an EC2 m5.12xlarge, Firecracker snapshot restore with the file backend takes as little as 4 ms — the dominant cost is not KVM setup but the latency of the first vCPU run after the memory mapping is established. Application-level cold start varies by runtime, from 20 ms to over 60 seconds unoptimized; the snapshot eliminates the portion spent in initialization.

The UFFD backend enables a further extension: transparent pre-warming. A handler that predicts the guest's access pattern can issue `UFFDIO_COPY` for likely-needed pages before the vCPU faults on them, masking fetch latency entirely.

Security Hazards

A snapshot is a frozen moment in a running machine's life. When that moment is resumed into multiple clones, every piece of state that looked unique at snapshot time is suddenly shared. The hazards are not theoretical: the arXiv paper by Brooker et al. (arXiv:2102.12892, 2021) documented successful TLS 1.0 attacks via snapshot reuse and catalogued five categories of cloned state: cryptographic key material, RNG seeds, session tokens, nonces, and UUID generators.

Entropy and the CSPRNG

The guest kernel's CSPRNG state is identical across all clones at the moment of restore. Divergence happens only through post-resume noise: timer jitter, interrupt timing, hardware RNG output. Until that noise accumulates, clones can produce identical key material, nonces, and session tokens.

Firecracker addresses this with **VMGenID**, an ACPI device (`_HID = "VMGENCTR"`, `_CID = "VM_Gen_Counter"`) that exposes a 128-bit generation counter in guest-physical memory. Firecracker added VMGenID support in v1.8.0 on x86_64. On every snapshot restore, Firecracker does three things: generates a fresh 128-bit value via `aws_lc_rs::rand::fill()`, writes it as little-endian bytes into the pre-allocated guest memory region, and triggers a Global System Interrupt to notify the guest. The ACPI path is `_SB_.VGEN`. The implementation lives in `src/vmm/src/devices/acpi/vmgenid.rs`.

The Linux VMGenID driver (`drivers/virt/vmgenid.c`) was merged in kernel 5.18 on x86_64. It detects the ID change and calls `add_vmfork_randomness()`, which calls `crng_reseed(NULL)` if the CSPRNG is already initialized, mixing in CPU HWRNG output (RDSEED/RDRAND) where available. The kernel logs `"crng reseeded due to virtual machine fork"` to confirm the reseed happened. On aarch64, the DeviceTree VMGenID binding arrived in kernel 6.10 — Firecracker supports guest kernels up to 6.1 on that architecture, so ARM users require a backported vmgenid patch.

VMGenID does not eliminate the hazard; it reduces the window. A **race window** exists between vCPU resumption and the VMGenID interrupt being processed. During that window, clones may produce identical output. Measuring the exact size of that window requires knowing when the guest kernel scheduler first runs the VMGenID driver's interrupt handler relative to any userspace code that calls `getrandom(2)`.

For guest kernels older than 5.18 (or ARM kernels without the vmgenid backport), reseed must be done manually. The guest-side procedure requires `CAP_SYS_ADMIN`:

> ***Note:*** *The following ioctls require root inside the guest and a file descriptor opened on* `/dev/random`.

Open `/dev/random`. Issue `RNDADDENTROPY` (`_IOW('R', 0x03, int[2])`, defined in `include/uapi/linux/random.h`) to mix entropy bytes into the input pool. Then issue `RNDRESEEDCRNG` (`_IO('R', 0x07)`, also requiring `CAP_SYS_ADMIN`) to force an immediate CSPRNG reseed.

VMGenID does not reach userspace PRNGs: OpenSSL's DRBG, Go's `math/rand`, Java's `SecureRandom`, and similar language-level generators cache entropy and are not notified by the kernel reseed event. Applications that use those generators must implement their own clone-detection and reseed logic.

Three additional frozen items require attention per clone:

`/proc/sys/kernel/random/boot_id` is a UUID written at boot time and never changed. Some libraries treat it as a per-instance unique ID. On a clone, all instances share the same `boot_id`. The mitigation is to bind-mount a freshly generated UUID file over it after restore.

`/var/lib/systemd/random-seed` is read by systemd at startup to reseed the kernel pool. If all clones share the same seed file from the base image, they all replay the same initial seed. The mitigation is to delete the file before taking the base snapshot, so no seed file is replayed.

Userspace PRNG state cannot be addressed at the hypervisor level at all. Each application that holds cached entropy must detect the clone event — via the `boot_id` change or a side-channel agreed on at clone startup — and reseed independently.

Frozen Clocks

The guest kvmclock freezes at snapshot time. On restore, the clock resumes from the frozen value; wall-clock time spent between snapshot and restore is invisible to the guest.

The kvmclock ABI exposes this discontinuity explicitly. Each vCPU communicates a `pvclock_vcpu_time_info` struct to KVM via `MSR_KVM_SYSTEM_TIME_NEW` (defined in `arch/x86/include/uapi/asm/kvm_para.h`). The `PVCLOCK_GUEST_STOPPED` flag bit (`0x2`) in that struct's `flags` field signals that the vCPU was suspended. KVM sets this bit on restore, and the guest kernel (`arch/x86/kernel/kvmclock.c`) reads it to detect time discontinuities. Guest code that cares about monotonic time can check this flag and compensate; code that does not check it will see an apparent time jump at the first tick after resume.

Firecracker v1.16.0 changed the default restore behavior: prior to that release, Firecracker unconditionally advanced the kvmclock by the elapsed wall time, which caused the monotonic clock to jump forward abruptly on kernels >= 5.16. The fix changed the default to leave the kvmclock frozen at the snapshotted value. The `clock_realtime` field on `PUT /snapshot/load` (x86_64 only) opts back into the old behavior: when `clock_realtime: true`, Firecracker passes the `KVM_CLOCK_REALTIME` flag (`1 << 2 = 0x4`) in the `flags` field of the `kvm_clock_data` struct issued via `KVM_SET_CLOCK` (`_IOW(KVMIO, 0x7b, struct kvm_clock_data)`, defined in `include/uapi/linux/kvm.h`), and KVM advances the guest kvmclock by the elapsed duration before the vCPUs start running.

Duplicated Tokens and Network State

Cloning is fundamentally incompatible with session-layer protocols that assume a single unique client identity. Firecracker closes vsock connections open at snapshot time when the snapshot is restored — the guest will observe a connection reset. Vsock listen sockets survive, with their CIDs updated. TCP connections inside the guest are not managed by Firecracker; they remain in the guest's network stack, where they will either time out or receive resets from the peer depending on whether the peer survived the snapshot interval.

The more serious problem is application-layer state cached in memory before the snapshot: TLS session tickets, OAuth tokens, signed cookies, HMAC keys, database connection credentials. Restoring the snapshot into two clones creates two clients presenting the same credentials to the same backends. Firecracker's `network_overrides` field remaps host tap device names between clones; it does not remap anything inside the guest.

The mitigations are application-level. The clone must either avoid caching credentials before the snapshot point, or must invalidate and re-acquire them as part of its resume initialization. The Lambda model solves this by snapshotting before the handler function is invoked — the function has not yet obtained any per-invocation credentials — and issuing fresh credentials to each clone via instance metadata.

Compatibility Constraints

Snapshot portability is more constrained than it appears. The magic ID enforces architecture separation: an x86_64 snapshot cannot be loaded on aarch64. Cross-CPU-vendor restore — AMD snapshot on an Intel host, or the reverse — is warned against via `get_vendor_id_from_host()` but not blocked;

behavior depends on which instructions and MSRs were captured in the CPUID and MSR state. Cross-kernel-version restore is described in the documentation as "considered unstable," with one known-tested pair: Linux 5.10 host to 6.1 host on m5n, m6i, and m6a EC2 instance types.

On aarch64, the GIC (Generic Interrupt Controller) version becomes a constraint: snapshots cannot be restored across different GIC versions, because the interrupt controller state embedded in `VmState` is GIC-version-specific.

cgroups V1 causes unexpectedly high snapshot restoration latency. The documentation recommends cgroups V2 strongly. The mechanism is that cgroups V1's per-subsystem hierarchy introduces extra synchronization during process creation and `mmap` at restore time; V2's unified hierarchy avoids this.

The snapshot format MAJOR version must match between the snapshot file and the running Firecracker binary. Format version `11.0.0` was current as of June 2026. The `versionize` crate used in Firecracker's development preview period was archived on 2026-02-27 and no longer appears in the codebase.

Sources And Further Reading

- Firecracker snapshot support documentation: https://github.com/firecracker-microvm/firecracker/blob/main/docs/snapshotting/snapshot-support.md
- Firecracker snapshot versioning documentation: https://github.com/firecracker-microvm/firecracker/blob/main/docs/snapshotting/versioning.md
- Handling page faults on snapshot resume: https://github.com/firecracker-microvm/firecracker/blob/main/docs/snapshotting/handling-page-faults-on-snapshot-resume.md
- Random number generation for clones: https://github.com/firecracker-microvm/firecracker/blob/main/docs/snapshotting/random-for-clones.md
- `src/vmm/src/snapshot/mod.rs` (magic IDs, `Snapshot<Data>`, CRC logic): https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/snapshot/mod.rs
- `src/vmm/src/persist.rs` (`SNAPSHOT_VERSION`, `MicrovmState`, create/restore entry points): https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/persist.rs
- `src/vmm/src/vmm_config/snapshot.rs` (`SnapshotType`, `LoadSnapshotParams`, `CreateSnapshotParams`, `MemBackendType`): https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/vmm_config/snapshot.rs
- `src/vmm/src/arch/x86_64/vcpu.rs` (`save_state()`, `restore_state()` ioctl sequences): https://github.com/firecracker-microvm/firecracker/tree/main/src/vmm/src/arch/x86_64/vcpu.rs
- `src/vmm/src/vstate/memory.rs` (`dump()`, `dump_dirty()`, memory file layout): https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/vstate/memory.rs
- `src/vmm/src/devices/acpi/vmgenid.rs` (VMGenID device implementation): https://github.com/firecracker-

microvm/firecracker/blob/main/src/vmm/src/devices/acpi/vmgenid.rs

- Firecracker CHANGELOG: https://github.com/firecracker-microvm/firecracker/blob/main/CHANGELOG.md
- KVM API documentation: https://docs.kernel.org/virt/kvm/api.html
- `include/uapi/linux/kvm.h` (`kvm_clock_data`, `KVM_CLOCK_REALTIME`, `KVM_SET_CLOCK`): https://github.com/torvalds/linux/blob/master/include/uapi/linux/kvm.h
- `include/uapi/linux/random.h` (`RNDADDENTROPY`, `RNDRESEEDCRNG`): https://github.com/torvalds/linux/blob/master/include/uapi/linux/random.h
- `drivers/virt/vmgenid.c` (Linux VMGenID driver, kernel 5.18+): https://github.com/torvalds/linux/blob/master/drivers/virt/vmgenid.c
- `drivers/char/random.c` (`add_vmfork_randomness`, `crng_reseed`): https://github.com/torvalds/linux/blob/master/drivers/char/random.c
- `arch/x86/include/asm/pvclock-abi.h` (`pvclock_vcpu_time_info`, `PVCLOCK_GUEST_STOPPED`): https://github.com/torvalds/linux/blob/master/arch/x86/include/asm/pvclock-abi.h
- `arch/x86/include/uapi/asm/kvm_para.h` (`MSR_KVM_SYSTEM_TIME_NEW`): https://github.com/torvalds/linux/blob/master/arch/x86/include/uapi/asm/kvm_para.h
- `arch/x86/kernel/kvmclock.c` (kvmclock resume behavior): https://github.com/torvalds/linux/blob/master/arch/x86/kernel/kvmclock.c
- `userfaultfd(2)` man page: https://man7.org/linux/man-pages/man2/userfaultfd.2.html
- Kernel userfaultfd admin guide: https://docs.kernel.org/admin-guide/mm/userfaultfd.html
- Brooker et al., "Restoring Uniqueness in MicroVM Snapshots," arXiv:2102.12892 (2021): https://arxiv.org/abs/2102.12892
- AWS Lambda SnapStart announcement: https://aws.amazon.com/blogs/aws/new-accelerate-your-lambda-functions-with-lambda-snapstart/
- AWS Lambda SnapStart internals: https://aws.amazon.com/blogs/compute/under-the-hood-how-aws-lambda-snapstart-optimizes-function-startup-latency/
- versionize crate (archived 2026-02-27): https://github.com/firecracker-microvm/versionize

Chapter 17: MMDS — The MicroVM Metadata Service

A microVM boots knowing almost nothing about itself. The kernel image was baked ahead of time; the root filesystem is a generic snapshot; the init process has no way to ask the hypervisor which instance it is, what its role is, or what credentials it should use. Someone has to tell it. The question is how.

The obvious approach — pack the information into the boot environment as kernel command-line arguments or environment variables inside the initial RAM disk — breaks in any fleet where the same image runs across thousands of machines with different identities. A cloud instance cannot encode its own hostname, role, or ephemeral credentials at image-build time. AWS learned this the hard way building EC2, and their solution, the **Instance Metadata Service** at `169.254.169.254`, became the de-facto standard for injecting per-instance configuration into a running virtual machine without rebuilding the image. When Firecracker was designed to run Lambda functions at scale, the Firecracker team needed the same capability — but they needed it without adding a privileged listener on the host network, without a sidecar process, and without allowing the guest to modify its own metadata.

What they built is the **MicroVM Metadata Service**, or **MMDS**. It is a JSON key-value store embedded in the VMM process, served to the guest over a purpose-built HTTP stack that lives entirely inside `firecracker`, and written exclusively by the host over the Unix-domain API socket. The guest reads; the host writes. That asymmetry is load-bearing: it is the entire security property of the design.

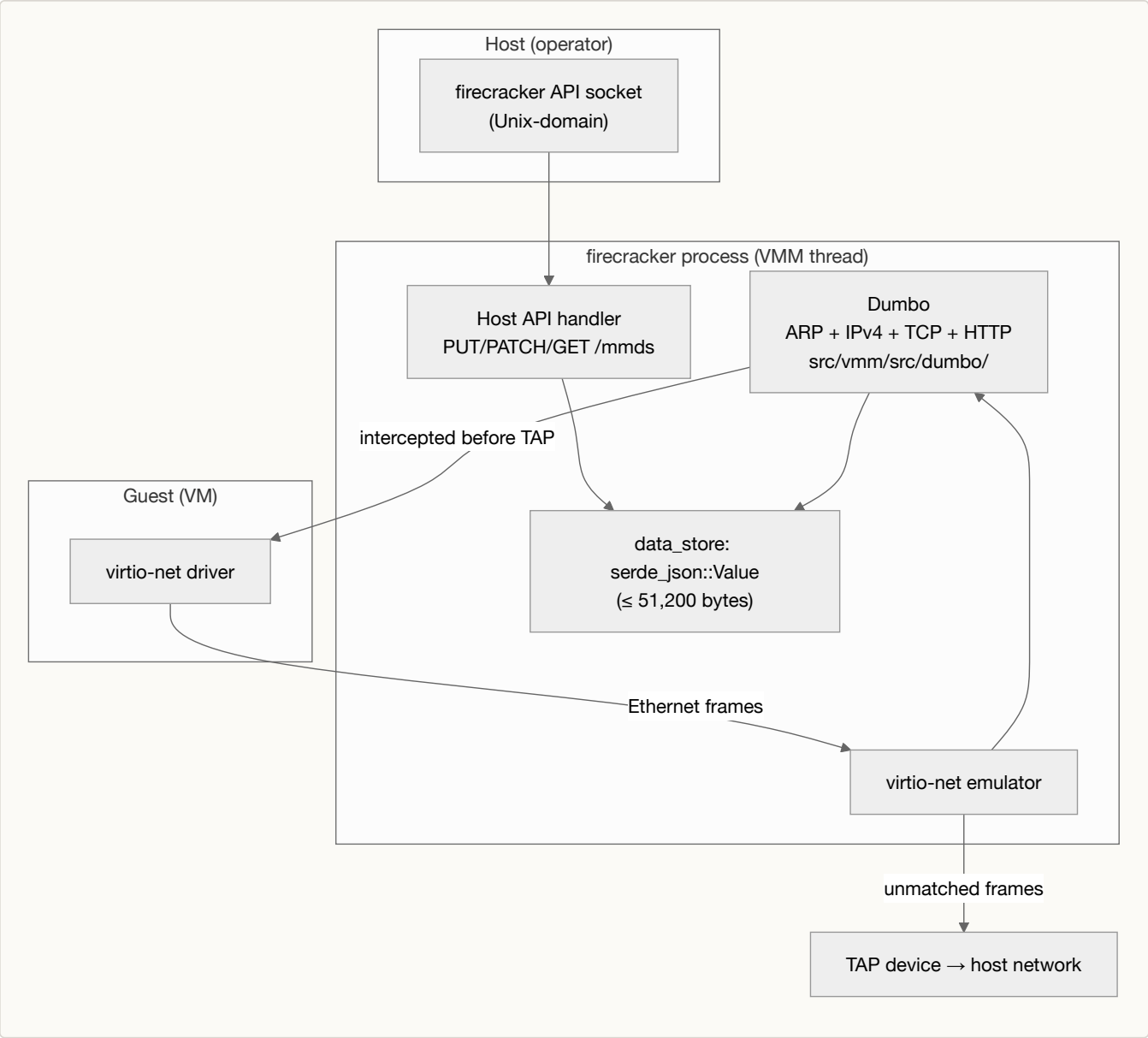
Where MMDS Lives

MMDS has three components, and all three run inside the single `firecracker` process — outside the KVM hardware boundary, in the VMM thread alongside the virtio-net, virtio-block, virtio-vsock, and virtio-balloon device emulators. There is no separate MMDS process, no sidecar, no additional thread.

The first component is the **host-side API handler**, which accepts `PUT /mmds`, `PATCH /mmds`, `GET /mmds`, and `PUT /mmds/config` from the operator over the Firecracker Unix-domain socket.

The second is the **data store**, a single `serde_json::Value` tree (`data_store` field on `struct Mmds` in `src/vmm/src/mmds/data_store.rs`) that holds whatever JSON the host has placed there. By default the store is capped at 51,200 bytes (50 KiB); the `--mmds-size-limit` CLI flag raises or lowers that ceiling, defaulting to the value of `--http-api-max-payload-size` when unset.

The third is **Dumbo**, a minimalist IPv4/TCP/HTTP network stack in `src/vmm/src/dumbo/`. Dumbo intercepts outbound Ethernet frames from the guest's virtio-net ring — before any of them reach the host TAP device — and synthesizes HTTP responses from the data store entirely in process.



The critical property visible in the diagram is the position of Dumbo: it sits between the guest's virtio-net ring and the TAP device, not between the TAP device and the host network. Frames that look like MMDS traffic never reach TAP. Frames that do not match pass through unmodified.

The Data Store

The data store is deliberately simple. It holds a single JSON tree — not a typed schema, not a relational model, just a `serde_json::Value`. The host populates it with `PUT /mmds` to replace the whole tree, or `PATCH /mmds` to apply a **JSON Merge Patch** (RFC 7396) against the existing contents. Under RFC 7396 semantics, a `null` value in the patch removes the corresponding key from the store; any non-null value replaces or creates it.

Only JSON objects and strings are supported when a guest reads them. Arrays, numbers, and booleans may exist in the store, but if the guest requests a path that resolves to one, Dumbo returns `501 Not Implemented` unconditionally — the `UnsupportedValueType` arm in `mod.rs` always maps to `StatusCode::NotImplemented` regardless of mode. When `imds_compat` is enabled, the response body is formatted as plain-text in IMDS style rather than a JSON envelope, but the `501` is not conditional on that flag.

The error variants on `MmdsDatastoreError` are: `DataStoreLimitExceeded`, `NotFound`, `NotInitialized`, `TokenAuthority(TokenError)`, and `UnsupportedValueType`. `NotInitialized` is returned when the guest queries the store before the host has written anything — the store starts empty, and an empty store is different from a store that has been explicitly populated with an empty object.

Configuring MMDS

Before the microVM boots, the operator sends a `PUT /mmds/config` request over the API socket to attach MMDS to one or more network interfaces and choose a version. The `MmdsConfig` struct (in `src/vmm/src/vmm_config/mmds.rs`, annotated `#[serde(deny_unknown_fields)]` — unknown fields return `HTTP 400`) has four fields:

Field	Type	Default
`version`	`V1` or `V2`	`V1` (deprecated)
`network_interfaces`	list of interface IDs	required
`ipv4_address`	IPv4 address	`169.254.169.254`
`imds_compat`	boolean	`false`

`PUT /mmds/config` is pre-boot only. The `network_interfaces` list names which of the guest's virtio-net devices will have a Dumbo instance attached. A guest interface not listed in `network_interfaces` has no associated Dumbo; its frames addressed to `169.254.169.254` pass through to TAP unmodified, which is why the production host-setup guide mandates firewall rules on every TAP interface regardless of configuration (see the trust model section below).

Sending `version: V1` — or omitting the field, since `V1` is the default — causes the API server to append the deprecation warning `"MmdsV1 is deprecated. Use V2 instead."` to the response body. V1 has been deprecated since Firecracker v1.1.0. The design doc does not commit to a specific removal version beyond noting it will happen in a future major release.

When `imds_compat` is `true`, all guest responses are formatted as plain-text in EC2 IMDS style regardless of the `Accept` header. This allows existing EC2 metadata clients inside the guest — tools that expect `169.254.169.254` to behave like AWS — to work without modification.

How Dumbo Intercepts Guest Traffic

Frame interception runs in `src/vmm/src/mmds/ns.rs`, in the function that processes each Ethernet frame coming off the virtio-net receive queue. Two lightweight speculative checks decide whether a frame belongs to MMDS:

For **ARP frames**, `test_speculative_tpa()` checks whether the Target Protocol Address matches the configured MMDS IPv4 (default `[169, 254, 169, 254]`, constant `DEFAULT_IPV4_ADDR` in `ns.rs`).

For **IPv4 frames**, `test_speculative_dst_addr()` checks whether the destination IP matches.

Frames that match either check are consumed by Dumbo. All others are forwarded to TAP unchanged. Non-TCP frames that match the MMDS IP — ICMP pings, UDP datagrams — are silently absorbed and counted in the `mmds.rx_accepted_unusual` metric; they get no response.

ARP

When a guest ARP request for the MMDS IP arrives, `detour_arp()` records the sender's MAC and IP, then synthesizes an ARP reply announcing the fake MMDS MAC address `06:01:23:45:67:01` (constant `DEFAULT_MAC_ADDR`). The host's real TAP MAC is never exposed. ARP replies are queued ahead of pending TCP segments in Dumbo's write pipeline, so the first HTTP request never stalls waiting for an unresolved ARP entry. ARP frames follow the standard 28-byte IPv4-over-Ethernet layout (`ETH_IPV4_FRAME_LEN = 28`): hardware type `0x0001`, operation `0x0002` for the reply.

TCP

Dumbo implements passive TCP only — it never opens a connection. The state machine is deliberately stripped of anything that is not needed for serving short HTTP requests to a cooperating guest:

- No `TIME_WAIT` state.
- No out-of-order segment buffering.
- No congestion control.
- No SACK, no TCP timestamps.
- A single duplicate ACK triggers immediate retransmission (not the conventional triple-dup-ACK threshold).
- Only the MSS option is parsed during the SYN/SYNACK handshake.

Key constants from the source:

Constant	Value	Source
`MSS_DEFAULT`	536 bytes (RFC 879 minimum)	`tcp/mod.rs`
`RCV_BUF_MAX_SIZE`	2,500 bytes	`tcp/endpoint.rs`
`CONNECTION_RTO_PERIOD`	1,200,000,000 cycles (~300 ms at 4 GHz)	`tcp/endpoint.rs`
`CONNECTION_RTO_COUNT_MAX`	15 retransmissions, then RST	`tcp/endpoint.rs`
`DEFAULT_TCP_PORT`	80	`ns.rs`
`DEFAULT_MAX_CONNECTIONS`	30	`ns.rs`

If the receive buffer fills before a complete HTTP request arrives — possible for unusually large request headers — the connection is reset. Initial sequence numbers are generated via `xor_pseudo_rng_u32()`; in fuzz builds they are fixed at `0x12345678` to make traces reproducible.

IPv4 packets emitted by Dumbo carry `TTL = 1` (`DEFAULT_TTL: u8 = 1` in `pdu/ipv4.rs`). IP options are not supported; the header is always the standard 20-byte form.

HTTP Methods

Dumbo accepts only `GET` and `PUT` from the guest. Any other method returns `405 Method Not Allowed` with the header `Allow: GET, PUT`. The only valid target for `PUT` is `/latest/api/token` (constant `PATH_TO_TOKEN` in `token.rs`); a `PUT` to any other path returns `404 Not Found`. There is no code path by which a guest `PUT` invokes `put_data()` or `patch_data()` on the store. The guest can issue a PUT, but it gets a session token back, never the ability to modify what it reads.

Guest HTTP status codes:

Code	Condition
`200 OK`	Successful GET or token PUT
`400 Bad Request`	Malformed headers, invalid TTL, bad URI
`401 Unauthorized`	Missing or invalid/expired token (V2 only)
`404 Not Found`	JSON pointer path not found; PUT to non-token path
`405 Method Not Allowed`	Any method other than GET or PUT
`413 Payload Too Large`	Data store size limit exceeded
`501 Not Implemented`	Unsupported value type (array, number, or boolean at the requested path)

URI normalization collapses consecutive `/` slashes iteratively until stable before resolving the path as a JSON pointer into the data store tree.

V1 and V2: The Session Token Model

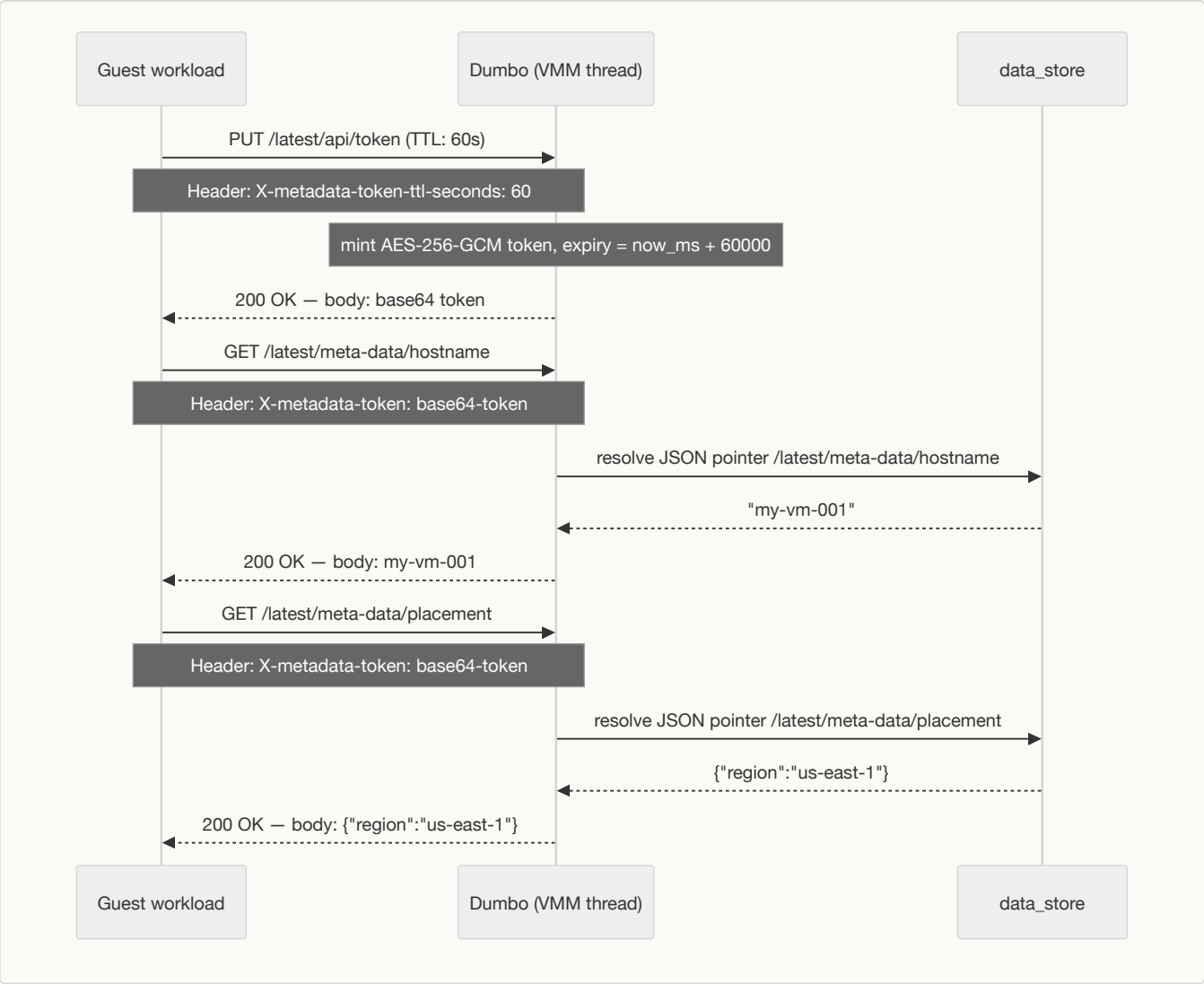
The fundamental difference between MMDS V1 and V2 is whether the guest must prove it is the legitimate tenant of the VM before reading metadata. V1 requires nothing. V2 requires a session token obtained from the MMDS itself before each batch of reads.

The threat V2 closes is **server-side request forgery** (SSRF). A guest workload — particularly a serverless function — might run code that an attacker controls. Without token gating, that code can `curl http://169.254.169.254/` and read the metadata unconditionally. With V2, the code still can — nothing prevents in-guest code from minting its own token — but the token is cryptographically bound to the specific `firecracker` instance, so it cannot be relayed to a different VM and used there.

V1 was the default from MMDS's introduction. V2 entered developer preview in Firecracker v0.23.0 and was promoted to GA in v1.1.0, at which point V1 was deprecated. The v1.0.0 release tightened V2 enforcement: session token validation was made mandatory for V2 GET requests, and `X-Forwarded-For` on any PUT to the token endpoint became an unconditional rejection.

Acquiring a V2 Token

The V2 flow is a three-step session protocol:



Step one: the guest sends `PUT http://169.254.169.254/latest/api/token` with the header `X-metadata-token-ttl-seconds: <N>`, where `<N>` is the requested lifetime in seconds. Firecracker v1.13.0 added aliases for EC2 IMDS compatibility: the equivalent EC2 header `X-aws-ec2-metadata-token-ttl-seconds: <N>` is also accepted. TTL bounds are enforced: minimum **1 second**, maximum **21,600 seconds** (six hours), defined as `MIN_TOKEN_TTL_SECONDS: u32 = 1` and `MAX_TOKEN_TTL_SECONDS: u32 = 21600` in `token.rs`. A TTL outside this range returns `400 Bad Request`.

Step two: Dumbo mints a token and returns it as a plaintext base64 string in the response body. The token response echoes back whichever TTL header variant the guest sent, matching EC2 IMDS behavior.

Step three: the guest presents the token on subsequent `GET` requests via `X-metadata-token: <token>` (or its EC2 alias `X-aws-ec2-metadata-token: <token>`). A missing or expired token in V2 mode returns `401 Unauthorized`.

The `X-Forwarded-For` header on any `PUT` to the token endpoint is unconditionally rejected with `400 Bad Request`. This blocks the simplest SSRF relay: a guest trying to obtain a token on behalf of a different host by routing the request through a forwarding proxy.

Under V1, both the `rx_no_token` and `rx_invalid_token` metrics are incremented when a token is absent or invalid, but the request proceeds normally with a `200 OK`. Under V2, the same metrics are incremented and the request returns `401`. These two metrics were added in Firecracker v1.13.0.

Token Cryptography

The token is not a random opaque nonce that Firecracker stores in a table. It is a self-contained sealed credential: anyone who holds the right key can verify it without a round-trip to a store. The structure (`struct Token`, `#[repr(C)]`, in `token.rs`) encodes everything needed for validation in 36 raw bytes:

Field	Length	Purpose
IV	12 bytes (`IV_LEN = 12`)	Per-token randomized nonce
Payload	8 bytes (`PAYLOAD_LEN = size_of::<u64>()`)	AES-256-GCM encrypted expiry timestamp
Tag	16 bytes (`TAG_LEN = 16`)	GCM authentication tag
Total	**36 bytes raw / 48 characters base64**	

The cipher is **AES-256-GCM** via the `aws-lc-rs` crate, using `RandomizedNonceKey::new(&AES_256_GCM, &key)`. The 256-bit key (`KEY_LEN: usize = 32`) is generated fresh when the `TokenAuthority` is created — once per microVM — and held entirely in the VMM process. The guest never sees it.

The payload is the expiry timestamp: `expiry_ms = now_ms + (ttl_seconds * 1000)`, where `now_ms` is the monotonic clock at mint time. To validate, Dumbo decrypts the payload and checks that `expiry_ms > current_monotonic_ms`. The GCM tag covers the ciphertext; forgery is computationally infeasible.

The Additional Authenticated Data (AAD) passed to AES-GCM is `"microvmid={instance_id}"`. This binds each token to the specific Firecracker instance that minted it. A token from one microVM presented to a different microVM fails GCM authentication because the AAD does not match; the tag is invalid.

Incoming tokens longer than 70 characters (`TOKEN_LENGTH_LIMIT = 70`) are rejected before any decryption attempt. The actual emitted token is always 48 characters (the base64 encoding of 36 bytes), so this limit provides a clean denial-of-service guard: a guest cannot cause Firecracker to do arbitrarily large decryption work by submitting a long string.

Key Rotation

`TokenAuthority` tracks how many tokens it has minted in `num_encrypted_tokens: u32`. When that counter reaches `u32::MAX` (4,294,967,295 tokens), the authority re-seeds with a fresh 256-bit random key, resets the counter to zero, and invalidates all previously issued tokens. In practice this threshold is unreachable: at one token per second it would take over 136 years per microVM. The rotation path exists because AES-GCM nonce reuse with the same key would be catastrophic; exhausting the nonce space — which `RandomizedNonceKey` is designed to detect — triggers safe regeneration rather than silent failure.

The Trust Model

The design document states the rule plainly:

> *"MMDS related API requests come from the host, which is considered a trusted environment, so there are no checks beside the kind of validation done by HTTP server and `serde-json`."*
> *"guest traffic should be treated as untrusted, and firewall rules should be put in place at the host-level to prevent guests from accessing restricted IPv4 addresses on the host."*

These two statements define the full trust boundary. The host has unconditional write access to the data store via the Unix-domain socket and faces no authentication challenge from MMDS. The guest has read access to whatever the host has placed there and cannot write or modify a byte of it.

What the Guest Cannot Do

The guest cannot reach the Firecracker API socket — that is a file on the host filesystem, inaccessible from inside the VM. The AES-256-GCM key used to mint and verify tokens exists only inside the VMM process; the guest cannot extract it. The host TAP device's real IP and MAC are never revealed: Dumbo's ARP reply gives the guest only the fake MAC `06:01:23:45:67:01`. The guest cannot discover data not explicitly placed in the store by the host, cannot determine whether non-TCP packets it sent to `169.254.169.254` were absorbed or dropped, and cannot cause Dumbo to modify the data store via any guest-accessible HTTP method.

Dumbo Is Not a Security Boundary

This point deserves stating directly because Dumbo looks like a firewall. It is not. Dumbo intercepts frames *when the guest uses the right interface and the right destination IP*. A guest interface not listed in `MmdsConfig.network_interfaces` has no Dumbo attached; if a guest sends an ARP request for `169.254.169.254` on that interface, the frame reaches the host TAP device and continues into the host network.

The production host-setup guide (`docs/prod-host-setup.md`) mandates host-level packet filtering to close this gap. Before running any Firecracker workload:

> ***Safety note:*** *the commands below modify host network filtering rules and require root privileges on the host.*

```
# nftables
nft add rule firecracker filter iifname "tap*" ip daddr 169.254.169.254 counter drop

# iptables-nft
iptables-nft -I FORWARD -i tap+ -d 169.254.169.254 -j DROP
```

These rules drop any frame arriving from a TAP interface that is destined for `169.254.169.254`, preventing a guest from reaching any host listener at that address regardless of how it routes the packet. Without them, a guest could potentially bypass Dumbo entirely and reach a host service at the link-local address.

How V2 Closes Cross-VM Token Relay

Consider the attack: a compromised guest A extracts a token from B's metadata service by somehow causing B to issue one and then forwarding it to A's MMDS endpoint. Because every `TokenAuthority` holds a distinct AES-256-GCM key and a distinct instance ID in the AAD, A's `TokenAuthority` cannot decrypt a ciphertext minted by B's key. The GCM tag fails verification and the token is rejected with `401 Unauthorized`. The `X-Forwarded-For` rejection on PUT adds a second layer: the simplest mechanical relay — routing the token PUT through an HTTP proxy — fails before the cryptographic check even runs.

Metrics

`MmdsMetrics` in `src/vmm/src/logger/metrics.rs` tracks MMDS health as `SharedIncMetric` counters flushed on read:

Metric	Meaning
`rx_accepted`	Frames rerouted to MMDS
`rx_accepted_err`	Errors handling a rerouted frame
`rx_accepted_unusual`	Non-TCP frames destined for MMDS IP (consumed, no response)
`rx_bad_eth`	Frames not parseable as Ethernet
`rx_invalid_token`	GET with invalid or expired token
`rx_no_token`	GET with no token
`rx_count`	Total successful receives
`tx_bytes`	Total bytes sent to guest
`tx_count`	Total successful sends
`tx_errors`	Send errors
`tx_frames`	Total frames sent
`connections_created`	TCP connections accepted by Dumbo
`connections_destroyed`	TCP connections cleaned up

In V1 mode, `rx_invalid_token` and `rx_no_token` increment but the request succeeds. In V2 mode the same increment accompanies a `401`. Distinguishing the two via metrics is how an operator can measure V1 traffic patterns before migrating to V2.

Snapshot and Restore

Firecracker's snapshot mechanism persists `MmdsNetworkStackState` (in `src/vmm/src/mmds/persist.rs`): the MAC address, the IPv4 address as a `u32`, the TCP port as a `u16`, the MMDS version, and the network interface configuration. What it does not persist is the data store contents or the token authority. On restore, a fresh `TokenAuthority` is created with a new random key, which immediately invalidates any V2 tokens that were valid at snapshot time. The data store starts empty. Operators must re-populate the store via `PUT /mmds` after restoring a snapshot — this is an explicit requirement in the user guide, not an implementation detail.

This is a natural consequence of the data store living as an in-memory `serde_json::Value`. It was never written to disk, so there is nothing to restore. Keeping TCP connection state alive across a snapshot/restore without replaying the data store is the point: network identity persists so the guest's connections do not stall; the host re-provides content deliberately, not by accident.

Version History

Firecracker version	Change
v0.22.0	`PUT /mmds/config` pre-boot endpoint added; `allow_mmds_requests` per-interface flag removed from the network interface API
v0.23.0	MMDS V2 introduced as developer preview via optional `version` field on `PUT /mmds/config`; default remains V1
v1.0.0	V2 session token enforcement made mandatory for GET requests; `X-Forwarded-For` on `PUT /latest/api/token` rejected unconditionally regardless of header casing
v1.1.0	MMDS V2 promoted to GA; V1 deprecated; `--mmds-size-limit` CLI flag added; MMDS version persisted in snapshots
v1.13.0	`imds_compat` field added; AWS EC2 header aliases added; `rx_invalid_token` and `rx_no_token` metrics added; token response echoes TTL header

The asymmetric read/write model — host writes, guest reads, no exceptions — is what lets fleet operators inject arbitrary per-instance configuration without trusting the workload. Chapter 18 examines the other side of that relationship: how the guest signals intent back to the host via `virtio-vsock`, and when an out-of-band channel like that is worth the complexity.

Sources and Further Reading

- MMDS design document — architecture, Dumbo design rationale, and the primary trust model statements quoted in this chapter.
- MMDS user guide — operator-facing V1/V2 workflow, `Accept` header semantics, snapshot caveats.
- `src/vmm/src/mmds/data_store.rs` — `Mmds` struct, `MmdsDatastoreError`, size limit, RFC 7396 patch semantics.
- `src/vmm/src/vmm_config/mmds.rs` — `MmdsConfig` struct, `deny_unknown_fields`, `imds_compat`.
- `src/firecracker/src/api_server/request/mmds.rs` — host API handler, V1 deprecation message text.
- `src/vmm/src/mmds/mod.rs` — `convert_to_response()`, V1/V2 GET dispatch, `PATH_TO_TOKEN`, HTTP status codes.
- `src/vmm/src/mmds/token.rs` — `TokenAuthority`, AES-256-GCM via `aws-lc-rs`, `IV_LEN = 12`, `PAYLOAD_LEN = 8`, `TAG_LEN = 16`, `TOKEN_LENGTH_LIMIT = 70`, TTL bounds, key rotation on `u32::MAX`.
- `src/vmm/src/mmds/token_headers.rs` — header name constants (Firecracker native and AWS EC2 aliases).

- `src/vmm/src/mmds/ns.rs` — Dumbo frame interception, `DEFAULT_IPV4_ADDR`, `DEFAULT_MAC_ADDR`, `DEFAULT_TCP_PORT`, ARP reply synthesis.
- `src/vmm/src/mmds/persist.rs` — `MmdsNetworkStackState` snapshot fields.
- `src/vmm/src/dumbo/tcp/connection.rs` — Dumbo TCP passive state machine, ISN generation via `xor_pseudo_rng_u32()`.
- `src/vmm/src/dumbo/tcp/endpoint.rs` — `RCV_BUF_MAX_SIZE = 2500`, `CONNECTION_RTO_PERIOD`, `CONNECTION_RTO_COUNT_MAX = 15`.
- `src/vmm/src/dumbo/tcp/mod.rs` — `MSS_DEFAULT = 536`, `MAX_WINDOW_SIZE = 1,073,725,440`.
- `src/vmm/src/dumbo/pdu/ipv4.rs` — `DEFAULT_TTL = 1`, standard 20-byte header enforcement.
- `src/vmm/src/dumbo/pdu/arp.rs` — `ETH_IPV4_FRAME_LEN = 28`, hardware type and operation constants.
- `src/vmm/src/logger/metrics.rs` — `MmdsMetrics` struct and all counter names.
- Firecracker design document — VMM thread model, device set, MMDS placement in the process.
- Production host setup guide — mandatory nftables and iptables-nft rules.
- Firecracker CHANGELOG — version history for V2 developer preview (v0.23.0), V2 token enforcement and X-Forwarded-For rejection (v1.0.0), V2 GA and V1 deprecation (v1.1.0), imds_compat and EC2 aliases (v1.13.0).

PART V — SECURITY AND ISOLATION

Chapter 18: The Jailer

The hardware virtualization boundary is real, but it is not the outermost wall. Once a process holds an open file descriptor on `/dev/kvm`, the kernel imposes no further restriction on which KVM ioctls that process can issue. A guest that breaks out of its vCPU execution environment and controls the VMM process inherits the entire ioctl surface. The question, then, is what limits the blast radius of a compromised VMM. The answer for Firecracker is `jailer` — a small privileged binary that wraps every Firecracker instance in the same Linux isolation primitives that contain an OCI container: a `pivot_root`-based filesystem jail, mount and PID namespaces, cgroup resource limits, rlimit caps, and a uid/gid privilege drop. The jailer is not itself a VMM; it never touches `/dev/kvm`. It exists to impose a second boundary around the process that does.

Why the Hardware Boundary Is Not Enough

KVM has had exploitable bugs. CVE-2021-29657, fixed in Linux 5.11.12, is a use-after-free in the SVM nested-virtualization path (`arch/x86/kvm/svm/nested.c`, function `nested_svm_vmrun()`) allowing an AMD KVM guest to bypass MSR access controls on the host (the VMCB12 double-fetch in `nested_svm_vmrun()` is one proposed mechanical account; the NVD record classifies it as CWE-416). The virtio 1.2 spec designates the split virtqueue descriptor table (Section 2.7.5) as guest-writable memory; device backends that trust descriptor contents without bounds-checking have produced heap overflows reachable from the guest — CVE-2019-14835, in the Linux kernel's `vhost` subsystem (`vhost/vhost.c`), exploited `get_indirect()` and was fixed in Linux 5.3.

Neither of these is a flaw in Firecracker. But they are evidence of a pattern: every abstraction between guest code and host resources has had bugs, and the hardware boundary is only one layer. Firecracker's `docs/design.md` states the position plainly: "all vCPU threads are considered to be running malicious code as soon as they have been started; these malicious threads need to be contained." The jailer is the outermost layer of that containment. It does not prevent the bugs from existing; it limits what a successful exploit can reach.

One Binary, One Instance, One Jail

Starting as root is a cost. `jailer` is a single Rust binary, distributed alongside `firecracker` in the same release tarball, whose entire purpose is to pay that cost once and then eliminate the privilege. It performs every privileged operation required to build and enter a jail, drops to a caller-specified uid and gid, and replaces itself with `firecracker` via `execve`. After the exec, `firecracker` never had root; it has never seen a file path outside the jail; it is constrained by cgroup limits that were set before it existed. Every Firecracker instance runs inside its own jail. Each jail is an independent directory tree, an independent set of cgroup leaves, and a distinct uid/gid pair — so a process that escapes one instance cannot reach another instance's files.

The minimum required arguments are:

Argument	Meaning
`--id <id>`	Unique identifier for this microVM instance
`--exec-file <path>`	Path to the `firecracker` binary on the host
`--uid <uid>`	uid the jailed process will run as
`--gid <gid>`	gid the jailed process will run as

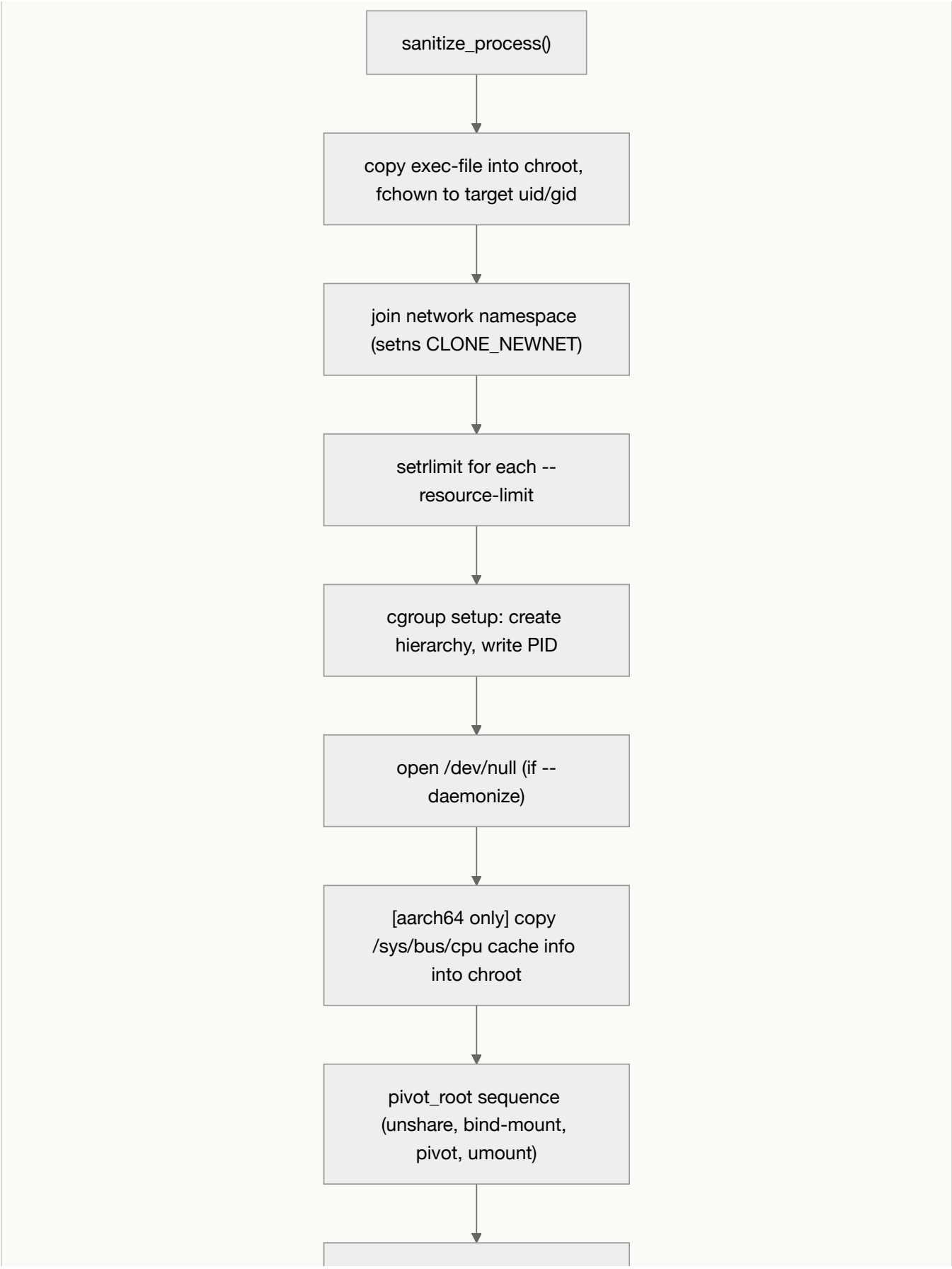
Additional flags wire in a network namespace (`--netns`), set cgroup limits (`--cgroup`, `--cgroup-version`, `--parent-cgroup`), add rlimit overrides (`--resource-limit`), daemonize the process (`--daemonize`), and place the process in a fresh PID namespace (`--new-pid-ns`). The chroot base directory defaults to `/srv/jailer`. Given `--exec-file /usr/bin/firecracker --id i1`, the jail root is `/srv/jailer/firecracker/i1/root` — constructed as `<chroot-base-dir>/<exec-file-name>/<id>/root`. That path is the filesystem root `firecracker` sees when it starts.

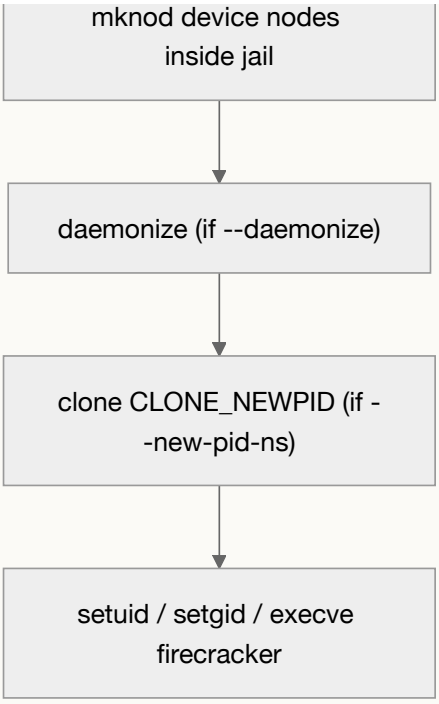
Process Sanitization

Before parsing arguments, `sanitize_process()` in `main.rs` closes every file descriptor from 3 upward by calling `close_range(3, UINT_MAX, CLOSE_RANGE_UNSHARE)` — Linux syscall `SYS_close_range`, available since kernel 5.9. It then removes every environment variable from the process environment via `clean_env_vars()`. Both operations happen unconditionally, before anything else: the goal is to ensure that whatever the caller passed to `jailer` — inherited fds, proxy-authorization tokens in the environment, sockets, anything — does not survive into the jail. The jailer's own subsequent work starts from a clean slate.

The Ordered Sequence

The bulk of the work lives in `Env::run()` in `src/jailer/src/env.rs`. The ordering is not arbitrary; each step has a dependency on what comes before it, and the sequence must be understood as a unit.


sanitize_process()
copy exec-file into chroot, fchown to target uid/gid
join network namespace (setns CLONE_NEWNET)
setrlimit for each --resource-limit
cgroup setup: create hierarchy, write PID
open /dev/null (if --daemonize)
[aarch64 only] copy /sys/bus/cpu cache info into chroot
pivot_root sequence (unshare, bind-mount, pivot, umount)



The network namespace join happens before cgroup setup and before the filesystem root changes, because `--netns` is a host-side path that must be opened and passed to `setns(2)` while the jailer can still see the host filesystem. Cgroup setup must complete before the chroot step for the opposite reason: the cgroup pseudo-filesystem hierarchy under `/sys/fs/cgroup` is not visible inside the jail, so any write to a cgroup control file must happen while the jailer still has access to the host namespace. Opening `/dev/null` before the chroot is the same logic — `daemonize` needs to redirect stdio, and `/dev/null` is a host path.

The Network Namespace Handoff

The jailer does not create a network namespace. It joins an existing one. The caller — typically a container runtime or a higher-level orchestrator — is responsible for creating the tap device inside the network namespace before invoking `jailer`. When `--netns <path>` is supplied, the jailer opens the namespace file with `O_CLOEXEC`, calls `setns(fd, CLONE_NEWNET)`, and closes the file descriptor. Everything after this point — cgroup writes, the chroot, the exec — inherits the joined namespace. `firecracker` starts inside the network namespace and has no mechanism to escape it.

The `docs/jailer.md` documentation places the network namespace join after the chroot sequence. The source code in `env.rs` places it before cgroup setup and before the chroot. The source is authoritative; this is noted explicitly because a reader who checks the documentation will see a different order.

Resource Limits

Two `rlimit` resources are supported via `--resource-limit <name>=<value>`:

Name	rlimit constant	Default
`no-file`	`RLIMIT_NOFILE`	2048
`fsize`	`RLIMIT_FSIZE`	not set

For each limit, `setrlimit(2)` is called with both `rlim_cur` (the soft limit) and `rlim_max` (the hard limit) set to the same value. This is the critical detail: a process can normally raise its own soft limit up to the hard limit at any time, without privilege. Setting both to the same value removes that headroom and makes the limit permanent for the lifetime of the process. The default of 2048 open file descriptors is intentionally conservative — a Firecracker instance with two vCPUs and a handful of virtio devices holds well under 100 open fds.

Cgroup Resource Control

Cgroup setup runs before the chroot and writes the jailer's own PID into the cgroup, so every subsequent operation — including the `execve` into Firecracker — inherits the cgroup membership. The controller discovery process parses `/proc/mounts` at runtime using a regex that distinguishes v1 and v2 mounts by the presence of a `2` suffix on the filesystem type:

```
^([a-z2]*)[[:space:]](?P<dir>.*)[[:space:]]cgroup(?P<ver>2?)[[:space:]](?P<options>.*)
[[:space:]]0[[:space:]]0$
```

The `(?P<ver>2?)` group is empty for cgroup v1 and `"2"` for cgroup v2. The implementation is in `src/jailer/src/cgroup.rs`.

cgroup v1

With `--cgroup-version=1` (the default), each controller has its own hierarchy. The jailer creates a directory `<controller-mount>/<parent-cgroup>/<id>/` — where `--parent-cgroup` defaults to the exec-file name, `firecracker` — writes the requested values into the controller-specific files, and attaches the process by writing its PID to `<cgroup-path>/tasks`. The `tasks` file accepts thread IDs under v1; process IDs written there apply to the entire thread group.

The `cpuset` controller requires special handling. If the parent cgroup's `cpuset.mems` or `cpuset.cpus` files are empty, attaching a process fails. The jailer calls `inherit_from_parent()`, which walks up the cgroup hierarchy until it finds a non-empty value, then propagates it down. Without this, a freshly created cgroup under a default install would refuse to accept any process.

Production-relevant resource knobs documented in `docs/prod-host-setup.md`:

Category	cgroup v1 file	Meaning
Memory hard limit	`memory.limit_in_bytes`	Hard memory ceiling
Memory + swap limit	`memory.memsw.limit_in_bytes`	Combined memory and swap cap
Memory soft limit	`memory.soft_limit_in_bytes`	Sharing threshold under contention
CPU weight	`cpu.shares`	Relative CPU weight
CPU period	`cpu.cfs_period_us`	CFS period (default 100,000 µs)
CPU quota	`cpu.cfs_quota_us`	CPU time allowed per period
IO IOPS	`blkio.throttle.io_serviced`	IOPS throttle
IO throughput	`blkio.throttle.io_service_bytes`	Bandwidth throttle

cgroup v2

With `--cgroup-version=2`, there is a single unified hierarchy. Before writing any controller property, the jailer must enable each required controller by writing `+<controller>` to every `cgroup.subtree_control` file along the path from the mount root down to the parent cgroup. This is the no-internal-process constraint: a non-root cgroup can distribute domain resources to child cgroups only if it contains no processes itself. The jailer satisfies this constraint by calling `write_all_subtree_control()` before creating or touching any child directory. Once controllers are enabled, it creates `<v2-root>/<parent-cgroup>/<id>/` and writes the PID to `cgroup.procs` — not `tasks`. Writing to `cgroup.procs` migrates all threads of the process atomically.

The v1/v2 distinction here is not just a naming difference. In v1, `tasks` and `cgroup.procs` exist side by side and mean different things; in v2 only `cgroup.procs` exists and the subtree enablement step is mandatory. If a domain controller is already enabled in a cgroup that contains processes and you attempt to move a PID into a child, the kernel rejects the write. The jailer's error message for this case includes: "Hint: If you intended to create a child cgroup under {0}, pass any --cgroup parameters."

A note for Linux 6.1 (x86_64) deployments: a boot regression affecting cgroup v2 performance is mitigated by remounting the unified hierarchy with `favordynmods` — `mount -o remount,favordynmods /sys/fs/cgroup`. The `favordynmods` option is specific to the cgroup v2 unified mount; it does not apply to cgroup v1 per-controller mounts.

The pivot_root Sequence

The filesystem jail is implemented in `src/jailer/src/chroot.rs`. The function is named `chroot()` in the source, but it performs a `pivot_root(2)` sequence rather than a simple `chroot(2)` call. The distinction matters: a process holding `CAP_SYS_CHROOT` can escape a `chroot(2)` jail by `chdir`-ing

outside the jail root before the root changes. `pivot_root` plus `MNT_DETACH` completely unmounts the old filesystem tree from the process's view. This is the same choice that `runc` and `youki` make for OCI containers.

> ***Root required.*** *The following sequence requires `CAP_SYS_ADMIN` and `CAP_SYS_CHROOT`. In normal operation, `jailer` holds these capabilities as root before the chroot step; after the exec into Firecracker, they are gone. Do not test this sequence against a production system without understanding the mount namespace implications.*

The steps, in order:

1. `unshare(CLONE_NEWNS)` — creates a new mount namespace. Mount events inside the jail will not propagate to the host and vice versa.
2. `mount(NULL, "/", NULL, MS_SLAVE | MS_REC, NULL)` — sets mount propagation to `MS_SLAVE` recursively, ensuring that unmount events inside the jail cannot propagate to the host's mount namespace. The `docs/jailer.md` documentation says `MS_PRIVATE | MS_REC`; the source code uses `MS_SLAVE | MS_REC`. The source is authoritative.
3. `mount(<chroot_dir>, <chroot_dir>, NULL, MS_BIND | MS_REC, NULL)` — self bind-mounts the jail directory. `pivot_root(2)` requires the new root to be on a different mount point from the current root; the self bind-mount creates that requirement.
4. `mkdir("old_root")` inside the chroot directory, mode 0600.
5. `chdir(<chroot_dir>)`.
6. `libc::syscall(libc::SYS_pivot_root, ".", "old_root")` — raw syscall; there is no glibc wrapper for `pivot_root(2)`. The old root is now accessible at `./old_root`.
7. `chdir("/")` — `pivot_root(2)` does not reposition the current working directory.
8. `umount2("old_root", MNT_DETACH)` — lazily detaches the old root. Any process still holding a reference to it can continue, but no new path traversals reach the old filesystem.
9. `rmdir("old_root")` — removes the empty mountpoint.

After step 9, there is no path from inside the jail to the host filesystem. No `chroot(2)` call follows; the `pivot_root` sequence is complete.

Device Nodes Inside the Jail

After the chroot, the jailer creates the directory structure that Firecracker needs — `/dev`, `/dev/net`, and `/run`, each mode `0o700`, each `chown`'d to the target uid/gid — and then calls `mknod` to create the character device nodes that Firecracker will open. The major/minor values are derived from `Documentation/admin-guide/devices.txt`:

Device	Jail path	Major	Minor	Notes
KVM	`/dev/kvm`	10	232	misc device; required for all guest execution
TUN/TAP	`/dev/net/tun`	10	200	misc device; `/dev/net/` created first
urandom	`/dev/urandom`	1	9	failure is non-fatal; MMDS v2 unavailable without it
userfaultfd	`/dev/userfaultfd`	10	dynamic	minor number read from `/proc/misc` at runtime

Every node is created with mode `S_IFCHR | S_IRUSR | S_IWUSR` — character device, readable and writable only by the owner — and immediately `chown`'d to the target uid/gid. Two of these have non-trivial behavior. The `/dev/urandom` `mknod` is allowed to fail: if it fails, the jailer prints a warning (`"MMDS version 2 will not be available to use."`) and continues. The `userfaultfd` minor number is allocated dynamically by the kernel using `MISC_DYNAMIC_MINOR`; the jailer reads `/proc/misc`, finds the line containing `"userfaultfd"`, and parses the minor number from its first column. If the kernel was built without `userfaultfd` support, the entry is absent and the node is silently omitted.

PID Namespace

When `--new-pid-ns` is set, the jailer calls:

A null child stack is intentional — the child immediately replaces itself with `execve`, so there is no stack work to do in the interim. The jailer parent writes the child PID to `<exec-file-name>.pid` and exits. The child process becomes PID 1 inside the new namespace. This is the init role in Linux PID namespace semantics: orphaned processes created within the namespace are reparented to PID 1, and the namespace is destroyed when PID 1 exits.

The PID namespace provides isolation but also a monitoring handle. The orchestrator can watch the PID file and know exactly which host PID corresponds to the init of a given microVM instance.

Privilege Drop and the Handoff

The final operation before exec is the privilege drop. The jailer constructs a `std::process::Command` with `.uid(uid).gid(gid)` set. The Rust standard library applies these as `setuid(2)` and `setgid(2)` calls in the child process between `fork(2)` and `execve(2)`. The transition from uid 0 to an unprivileged uid causes the Linux kernel to drop all effective capabilities — including `CAP_SYS_ADMIN`, `CAP_NET_ADMIN`, and `CAP_SYS_CHROOT` — automatically on exec.

No explicit `capset` call appears in the jailer's privilege-drop path; the uid/gid transition is the mechanism by which effective capabilities are shed on exec. `prctl(PR_SET_NO_NEW_PRIVS, 1)` does not appear in the jailer either, but it must be issued inside Firecracker before seccomp filters are installed: `seccomp(2)` requires either `CAP_SYS_ADMIN` or the `no_new_privs` bit, and Firecracker holds neither after the uid drop. Chapter 19 covers where and how Firecracker sets that bit and installs per-thread filters.

The arguments passed to Firecracker on exec are:

The timing arguments allow Firecracker to report accurate startup metrics even though the clock starts before the exec. After this point, `docs/design.md` states: "past this point, Firecracker can only access resources that a privileged third-party grants access to (e.g., by copying a file into the chroot, or passing a file descriptor)." The gate closes at `execve`.

Isolation Per Instance

`docs/prod-host-setup.md` specifies that each Firecracker instance must use a distinct uid/gid pair. This is not a convention — it is load-bearing. Every file and directory created inside the jail, including the `firecracker` binary copy, all device nodes, and the jail root directory, was `chown`'d to the target uid/gid before the exec. A process that escapes its chroot will find itself owning only those files. A process that somehow escapes to the host filesystem will be an unprivileged user with a uid that no other instance shares — POSIX ownership prevents lateral movement to a neighboring instance's files.

```
flowchart LR
    host["Host filesystem"]
    j1["/srv/jailer/firecracker/i1/root\nuid 10001 / gid 10001"]
    j2["/srv/jailer/firecracker/i2/root\nuid 10002 / gid 10002"]
    j3["/srv/jailer/firecracker/i3/root\nuid 10003 / gid 10003"]
    host --> j1
    host --> j2
    host --> j3
```

Host-Level Concerns Beyond the Jailer

The jailer does not address every attack surface.

Instance Metadata Service filtering is the orchestrator's responsibility. Firecracker performs no network filtering. All guest egress reaches the host tap interface as untrusted traffic. Block guest access to the IMDS with a rule such as:

> ***Root required.*** *The following nftables rule modifies host network policy and takes effect immediately.*

```
nft add rule firecracker filter iifname "tap*" ip daddr 169.254.169.254 counter drop
```

SMT (simultaneous multithreading, or hyperthreading) creates cross-core speculative execution side channels — Spectre, MDS variants — that neither the hardware virtualization boundary nor the jailer prevents. In multi-tenant environments, SMT should be disabled at the host level.

KSM (Kernel Samepage Merging) deduplicates identical memory pages across VMs. The deduplication timing is observable as a memory-content oracle across VM boundaries. KSM must be disabled.

The `kvm-pit/<pid>` kernel thread that KVM creates for the PIT timer is not automatically placed in the microVM's cgroup. If strict resource accounting is required, an external agent must move it into the instance's cgroup after VM creation.

The guest can write to the serial device, which maps to host stdout or stderr, at an arbitrary rate. Production deployments should redirect serial output to a bounded buffer or `/dev/null` to prevent a guest from consuming host I/O capacity through the serial device.

The Full Picture

```
sequenceDiagram
    participant O as Orchestrator
    participant J as jailer (root)
    participant K as kernel
    participant F as firecracker (uid N)

    O->>J: exec jailer --id i1 --uid N --gid N ...
    J->>J: close_range(3, UINT_MAX, CLOSE_RANGE_UNSHARE)
    J->>J: clean_env_vars()
    J->>K: setns(netns_fd, CLONE_NEWNET)
    J->>K: setrlimit(RLIMIT_NOFILE, 2048)
    J->>K: write cgroup limits, write PID to tasks/cgroup.procs
    Note over J: tasks (v1) or cgroup.procs (v2)
    J->>K: unshare(CLONE_NEWNS)
    J->>K: mount MS_SLAVE | MS_REC
    J->>K: mount MS_BIND | MS_REC (chroot dir)
    J->>K: SYS_pivot_root, umount2 MNT_DETACH
    J->>K: mknod /dev/kvm, /dev/net/tun, /dev/urandom
    J->>K: clone(CLONE_NEWPID) (if --new-pid-ns)
    J->>K: setuid(N), setgid(N)
    J->>F: execve firecracker --id i1 ...
    Note over J: jailer exits
    F->>F: prctl(PR_SET_NO_NEW_PRIVS, 1), install seccomp filters per-thread
    F->>K: open /dev/kvm, KVM_CREATE_VM ...
```

Past the `execve`, `docs/design.md` states: "Firecracker can only access resources that a privileged third-party grants access to (e.g., by copying a file into the chroot, or passing a file descriptor)." The gate closes at exec.

Sources And Further Reading

- Jailer source, `Env::run()` and privilege drop: https://github.com/firecracker-microvm/firecracker/blob/main/src/jailer/src/env.rs
- `pivot_root` sequence: https://github.com/firecracker-microvm/firecracker/blob/main/src/jailer/src/chroot.rs
- Cgroup v1/v2 controller setup: https://github.com/firecracker-microvm/firecracker/blob/main/src/jailer/src/cgroup.rs
- Resource limits (`RLIMIT_NOFILE` default 2048): https://github.com/firecracker-microvm/firecracker/blob/main/src/jailer/src/resource_limits.rs
- Argument parser (required/optional flags, defaults): https://github.com/firecracker-microvm/firecracker/blob/main/src/jailer/src/main.rs
- Jailer user-facing documentation: https://github.com/firecracker-microvm/firecracker/blob/main/docs/jailer.md

- Production host setup (cgroup knobs, network filtering, SMT, KSM): https://github.com/firecracker-microvm/firecracker/blob/main/docs/prod-host-setup.md
- Firecracker design and threat model: https://github.com/firecracker-microvm/firecracker/blob/main/docs/design.md
- Seccomp per-thread filter architecture: https://github.com/firecracker-microvm/firecracker/blob/main/docs/seccomp.md
- `pivot_root(2)` man page: https://man7.org/linux/man-pages/man2/pivot_root.2.html
- `setns(2)` man page: https://man7.org/linux/man-pages/man2/setns.2.html
- `seccomp(2)` man page: https://man7.org/linux/man-pages/man2/seccomp.2.html
- Linux cgroup v2 kernel documentation: https://docs.kernel.org/admin-guide/cgroup-v2.html
- Linux device number registry: https://www.kernel.org/doc/html/latest/admin-guide/devices.html
- virtio 1.2 spec, split virtqueue descriptor table (Section 2.7.5): https://docs.oasis-open.org/virtio/virtio/v1.2/cs01/virtio-v1.2-cs01.html
- CVE-2021-29657 (KVM SVM nested-virtualization use-after-free, fixed Linux 5.11.12): https://nvd.nist.gov/vuln/detail/cve-2021-29657
- CVE-2019-14835 (vhost/vhost_net `get_indirect()` heap overflow, fixed Linux 5.3): https://www.openwall.com/lists/oss-security/2019/09/17/1

Chapter 19: Seccomp In Firecracker

Imagine a guest has found a bug in KVM. The hypervisor boundary held, but the guest is now executing arbitrary code inside a vCPU thread on the host. That thread is a regular Linux thread. Without further containment it can call any syscall the process is permitted to call: `socket`, `execve`, `fork`, `mount`. The kernel will evaluate each call on the host, in the context of the Firecracker process, with the firewall rules and network interfaces the host has configured. A guest that can issue syscalls on the host is a guest that has escaped.

This is the problem seccomp solves in Firecracker. Not by blocking the guest from breaking the KVM barrier — that is VT-x and SVM's job — but by making the prize on the other side of the barrier worthless. If a compromised vCPU thread can only call the twenty-four syscalls it actually needs, the host's attack surface from that thread shrinks to those twenty-four calls and nothing else.

Firecracker's design document makes this explicit: all vCPU threads are treated as running malicious code from the moment they start. The seccomp filter is not a defense of last resort. It is the expected containment path for a thread that has been taken over by a guest.

The Kernel Mechanism: `seccomp(2)` BPF

The tool Firecracker uses is `seccomp(2)` in filter mode (`SECCOMP_SET_MODE_FILTER`), available since Linux 3.17 as syscall number 317 on x86-64. A caller passes a classic BPF (cBPF) program to the kernel via this syscall; from that point on, every syscall entry for that thread runs the BPF program first. The program's return value decides the call's fate before the kernel even looks at the syscall number.

The BPF program receives a single argument: a pointer to a `struct seccomp_data` in the kernel's address space:

```
struct seccomp_data {
    int   nr;                   /* syscall number */
    __u32 arch;                 /* AUDIT_ARCH_* value */
    __u64 instruction_pointer; /* RIP at time of syscall */
    __u64 args[6];              /* up to 6 syscall arguments */
};
```

The `arch` field is not cosmetic. On x86-64, a 32-bit i386 compat call with `nr == 0` resolves to `restart_syscall`, whereas the same `nr` on the x86-64 native ABI resolves to `read`. A filter that checks `nr` without first confirming `arch == AUDIT_ARCH_X86_64` can be bypassed by routing a dangerous call through the wrong ABI. Firecracker's compiled filters always check the architecture first.

The return value from the BPF program is one of eight actions, in priority order from highest to lowest:

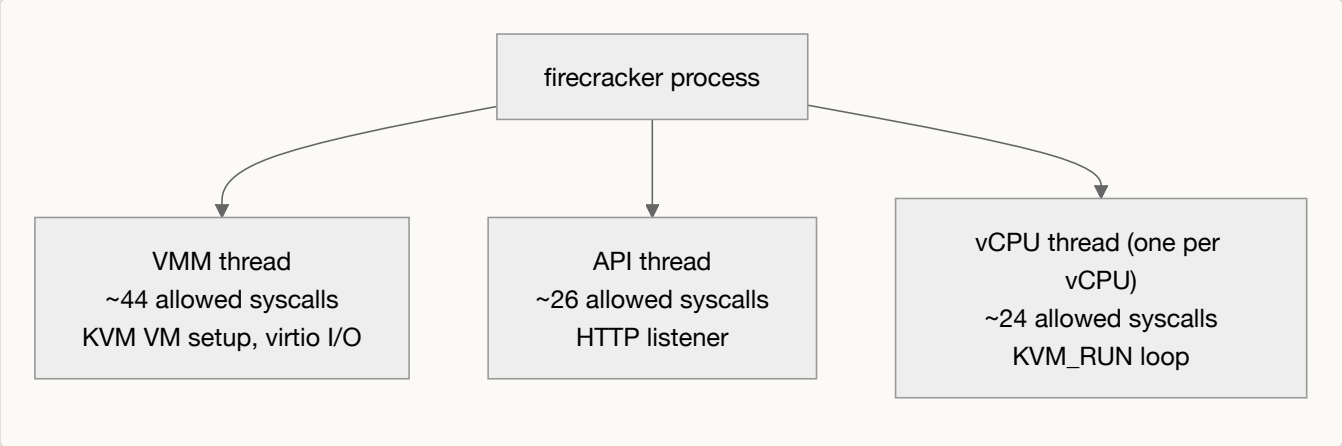
Action	Hex value	Effect
`SECCOMP_RET_KILL_PROCESS`	`0x80000000`	Kill entire process (Linux 4.14+)
`SECCOMP_RET_KILL_THREAD`	`0x00000000`	Kill calling thread
`SECCOMP_RET_TRAP`	`0x00030000`	Deliver `SIGSYS`
`SECCOMP_RET_ERRNO`	`0x00050000`	Return errno (low 16 bits)
`SECCOMP_RET_USER_NOTIF`	`0x7fc00000`	Notify userspace fd (Linux 5.0+)
`SECCOMP_RET_TRACE`	`0x7ff00000`	Notify ptrace tracer
`SECCOMP_RET_LOG`	`0x7ffc0000`	Allow and log (Linux 4.14+)
`SECCOMP_RET_ALLOW`	`0x7fff0000`	Permit syscall

When a thread has multiple chained filters — because `seccomp(2)` was called more than once — the kernel runs all of them and uses the action with the highest priority. The kernel enforces a hard cap of 4,096 BPF instructions per individual program (`BPF_MAX_LEN`) and 32,768 instructions across all chained filters for a single thread (`MAX_INSNS_PER_PATH`).

Before installing a filter, the calling thread must either hold `CAP_SYS_ADMIN` in its user namespace or have called `prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0)` . The no-new-privs flag prevents the thread and any process it exec-s from gaining privileges, so there is no privilege-escalation path the filter needs to worry about. Firecracker always takes this path; it never requires `CAP_SYS_ADMIN` to install its filters.

Three Threads, Three Filters

Firecracker runs as three categories of threads, each with a fundamentally different job and a fundamentally different syscall requirement:



The filter key for each thread — `"vmm"`, `"api"`, `"vcpu"` — maps to a separate BPF program compiled from the policy file at `resources/seccomp/x86_64-unknown-linux-musl.json`. None of the three programs is a subset of another; each is independently constructed to match what that thread actually calls.

The timing of filter installation matters. The VMM's filter goes in right before the event loop starts. The API thread's filter is installed right before the HTTP listener starts. The vCPU filter has the tightest constraint: the exact call chain is `start_threaded()` spawning a thread that calls `Vcpu::run(filter)`, which calls `apply_filter(filter)` before `StateMachine::run()` and therefore before any `KVM_RUN` ioctl reaches the kernel. The filter is in place before the first guest instruction executes.

If `apply_filter` fails on a vCPU thread, Firecracker panics immediately. There is no fallback and no soft failure. A vCPU that cannot install its seccomp filter is a vCPU that must not run.

From JSON Policy to Embedded BPF

The compilation pipeline runs at build time, not at Firecracker startup. The policy file is `resources/seccomp/x86_64-unknown-linux-musl.json`; the compiled output is an embedded binary blob in the Firecracker executable.

The JSON root is an object keyed on thread name. Each value is a filter configuration with three fields: `default_action`, which applies when no rule matches; `filter_action`, which applies when a rule matches; and `filter`, the array of allowed syscall rules. All three filters in the x86-64 default policy use `"default_action": "trap"` and `"filter_action": "allow"`. An unmatched syscall delivers `SIGSYS` to the Firecracker process, which its signal handler catches, logs, and converts to a controlled shutdown — more useful for post-incident analysis than `SECCOMP_RET_KILL`, which leaves no trace.

A rule entry is a JSON object with a `"syscall"` name and an optional `"args"` array:

```
{
  "vcpu": {
    "default_action": "trap",
    "filter_action": "allow",
    "filter": [
      { "syscall": "read" },
      {
        "syscall": "ioctl",
        "args": [
          { "index": 1, "type": "dword", "op": "eq", "val": 44672,
            "comment": "KVM_RUN" }
        ]
      }
    ]
  }
}
```

A rule without `"args"` is a **name-only** match: any invocation of that syscall passes. A rule with `"args"` is an **argument-level** match: every condition in the array must be true simultaneously (AND semantics), and multiple rules for the same syscall are ORed.

Each condition object has five fields:

Field	Type	Values
`index`	integer	0–5, the position in `args[6]`
`type`	string	`"dword"` (32-bit) or `"qword"` (64-bit)
`op`	string	`"eq"`, `"ge"`, `"gt"`, `"lt"`, `"le"`, `"ne"`, `"masked_eq"`
`val`	integer	base-10 decimal
`comment`	string	annotation only, ignored by compiler

The musl `ioctl` Quirk

When `"type": "dword"` is used with an `"eq"` comparison, the seccompiler does not emit a plain 64-bit equality check. It emits a `SCMP_CMP_MASKED_EQ` with the mask `0x00000000FFFFFFFF`, which zeroes the upper 32 bits before comparing. The reason: musl's `ioctl` wrapper leaves garbage in the upper 32 bits of the `request` argument. A plain 64-bit `eq` on `KVM_RUN` (decimal 44672, `0xAE80`) would reject legitimate calls whose upper bits happen to be non-zero. This is not a theoretical concern — it causes spurious `SIGSYS` signals in practice. The masking makes the comparison match what the kernel actually sees in `seccomp_data.args[1]`.

The Compilation Steps

1. Parse `x86_64-unknown-linux-musl.json` via `serde_json` into a `BTreeMap<String, FilterConfig>`. The use of `BTreeMap` rather than `HashMap` is deliberate: `HashMap` randomizes iteration order across builds, producing different BPF bytecode from identical inputs. PR #5298 fixed this by switching to `BTreeMap`, which guarantees deterministic key order and reproducible binaries.
2. For each thread key, call `seccomp_init(default_action)` to create a libseccomp context, then `seccomp_arch_add()` with the target architecture constant: `SCMP_ARCH_X86_64 = 0xc000003e` for x86-64 or `SCMP_ARCH_AARCH64 = 0xc00000b7` for ARM.
3. Resolve each syscall name to a number via `seccomp_syscall_resolve_name()`, then add it with `seccomp_rule_add()` for name-only rules or `seccomp_rule_add_array()` for rules with argument conditions.
4. Export the BPF bytecode via `seccomp_export_bpf()` to an anonymous `memfd` and read it back as `Vec<u64>`.
5. Serialize the full `BTreeMap<String, Vec<u64>>` with the `bitcode` crate and write the result to disk.

Firecracker v1.11.0 (PR #4926, merged 2025-01-16) migrated the compiler backend from a hand-written Rust BPF code generator to libseccomp. The migration reduced BPF program size by approximately 65%, which matters because a larger program means more instructions to evaluate at every syscall entry on every vCPU thread, every millisecond the guest runs.

Runtime Installation

At startup the Firecracker process reads the embedded bitcode blob and deserializes it. The deserialization is capped at `DESERIALIZATION_BYTES_LIMIT = 100,000` bytes to prevent an oversized custom filter file from triggering an unbounded allocation. The result is a map of thread name to `Vec<u64>` BPF programs.

Each thread's filter is installed by `apply_filter(bpf_filter: &[u64])` in `src/vmm/src/seccomp.rs`, which performs exactly four operations:

1. If `bpf_filter` is empty, return `Ok(())`. This path is used for debug builds and the `--no-seccomp` flag.
2. Call `prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0)`.
3. Construct a `SockFprog` with `len = bpf_filter.len() as u16` and `filter = bpf_filter.as_ptr()`.
4. Call `syscall(SYS_seccomp, SECCOMP_SET_MODE_FILTER, 0, &sock_fprog)`. Flags are zero: no `SECCOMP_FILTER_FLAG_TSYNC`, no `LOG`, no `NEW_LISTENER`.

BPF instructions are stored as `u64`, not as `sock_filter`. A BPF instruction is 8 bytes with 4-byte alignment; `u64` satisfies both constraints without requiring a custom struct at the call site. The kernel accepts the pointer regardless of which type the userspace caller uses.

The zero flag is intentional. `SECCOMP_FILTER_FLAG_TSYNC` (`1UL << 0`) would propagate the filter to all other threads in the process simultaneously; Firecracker does not use it. Instead it installs each thread's filter from within that thread, at the moment that thread reaches its operational steady state. The three BPF programs are different, so thread-sync installation would apply the wrong program to the wrong threads.

The x86-64 Default Allowlists

vCPU Thread: The Tightest List

The vCPU thread allows approximately 24 distinct syscalls. Its `ioctl` entry is not a single rule but 19 separate rules, each fixing `args[1]` to a specific request value. Selected entries from the compiled policy:

Syscall	Arg condition	Value	Comment
`ioctl`	`args[1] dword eq`	`44672`	`KVM_RUN`
`ioctl`	`args[1] dword eq`	`44707`	`KVM_GET_TSC_KHZ`
`ioctl`	`args[1] dword eq`	`44717`	`KVM_KVMCLOCK_CTRL`
`ioctl`	`args[1] dword eq`	`2147790488`	`KVM_GET_MP_STATE`
`ioctl`	`args[1] dword eq`	`2151722655`	`KVM_GET_VCPU_EVENTS`
`ioctl`	`args[1] dword eq`	`3221794440`	`KVM_GET_MSRS`
`ioctl`	`args[1] dword eq`	`3221794449`	`KVM_GET_CPUID2`
`mmap`	name-only	—	—
`futex`	name-only	—	—
`timerfd_settime`	name-only	—	—

The vCPU allowlist does not include `socket`, `connect`, `accept4`, `bind`, `fork`, `clone`, `execve`, `mount`, or any `io_uring_*` call. An `ioctl` with any request value outside the 19-entry list hits `default_action: trap`. The thread can run a guest, field exits, and adjust timers. It cannot open a network connection.

API Thread: Networking Without KVM

The API thread allows approximately 26 syscalls. Its purpose is to receive and respond to the Firecracker management API over a Unix socket, so it needs `accept4`, `recvfrom`, `recvmsg`, `sendto`, `socket`, `epoll_pwait`, and `getrandom`. It does not include `KVM_RUN` or any KVM VM-level ioctl. A compromised API thread can read and write on the management socket; it cannot trigger guest execution.

VMM Thread: The Broadest List

The VMM thread manages virtio device I/O, KVM VM setup, and the tap network interface, so its allowlist is the most permissive of the three: approximately 44 distinct syscalls. It includes `io_uring_enter`, `io_uring_setup`, and `io_uring_register` for io_uring-backed block devices; `connect`, `socket`, `sendmsg`, and `recvmsg` for the tap interface; and 14 distinct KVM ioctl request values covering VM-level operations such as `KVM_GET_DIRTY_LOG`, `KVM_SET_USER_MEMORY_REGION`, and `KVM_IRQFD`. Even at 44 calls, this is a fraction of the full Linux syscall table. The thread that orchestrates everything cannot, for instance, create a new process or modify the host's UID.

Name-Only vs. Argument-Level Filtering

The distinction between a name-only rule and an argument-level rule is not just a matter of precision — it is the difference between containing a class of calls and containing a specific operation.

A name-only `read` rule allows `read` on any file descriptor for any number of bytes. That is acceptable for a thread that should be able to read its event pipe, its timer fd, and its tap device fd — the policy author has decided that any `read` from this thread is legitimate. A name-only `ioctl` rule would be catastrophic: it would allow `KVM_RUN` and also `TIOCSTI` (inject bytes into a terminal), `PTRACE_TRACEME` encapsulated in `ioctl`, or any other driver-specific request the kernel understands.

The vCPU's `ioctl` entry is therefore 19 argument-level rules. Each one allows exactly one `request` value. Any `ioctl` whose second argument does not appear in that list — regardless of how the call is framed or what file descriptor it targets — hits `default_action: trap`. The argument filter makes the syscall permission specific enough to be meaningful.

The BPF cost of argument-level filtering is real. Each condition compiles to multiple BPF instructions: a load from the appropriate offset in `seccomp_data`, a masking step (for `dword` conditions), and a comparison. Nineteen conditions for `ioctl` alone is a substantial BPF program, and the total must stay under 4,096 instructions per filter. The 65% BPF size reduction from the libseccomp migration in v1.11.0 is what makes the current depth of argument filtering practical within that budget.

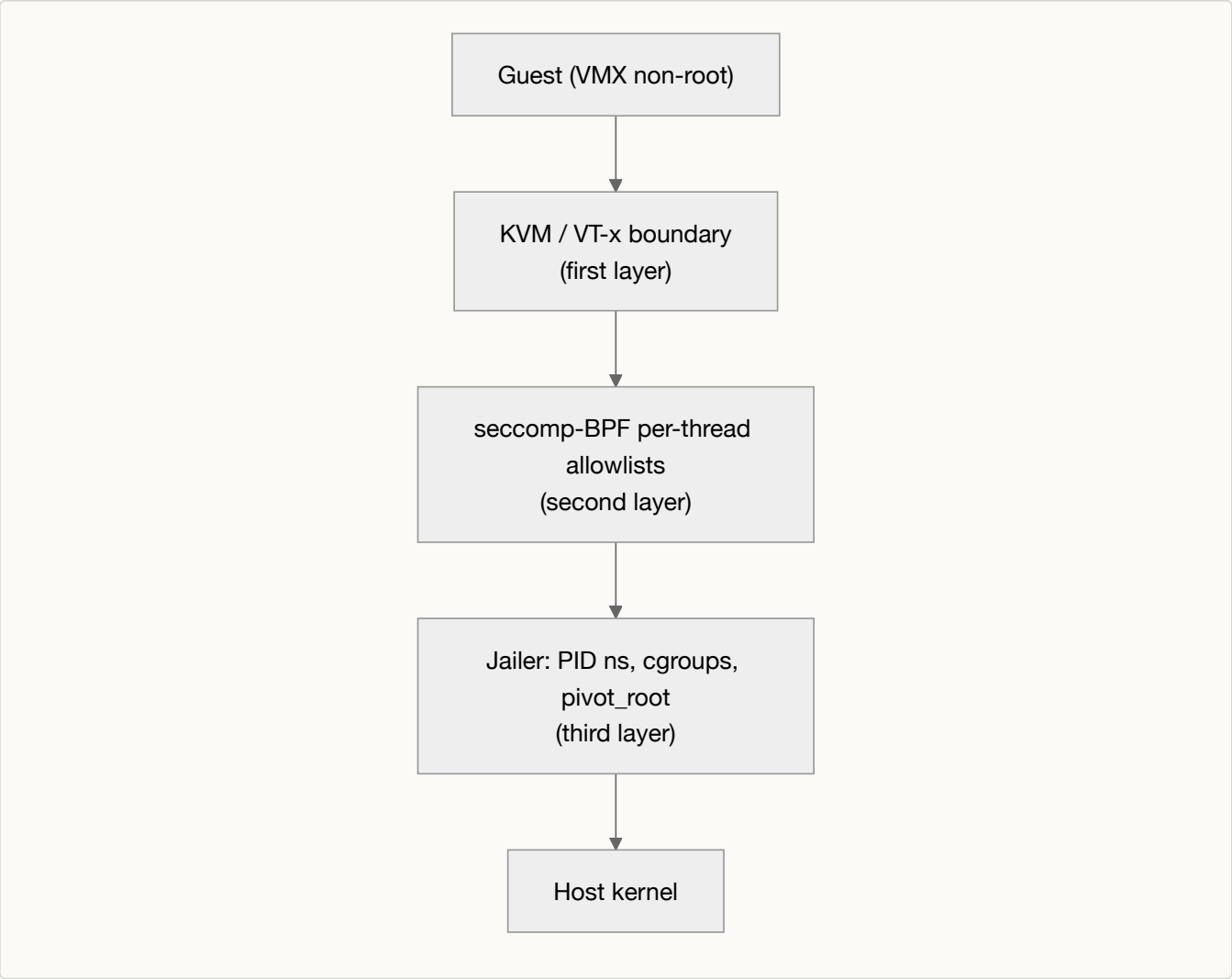
The `--basic` flag on `seccompiler-bin`, which stripped all argument conditions and produced name-only allowlists, was deprecated and removed. There is no longer a "basic mode" that silently downgrades policy precision; every compiled filter uses the same engine, and every argument condition in the JSON reaches the BPF program.

How Seccomp Shrinks the Host Attack Surface

Hardware virtualization is Firecracker's first containment layer. VT-x and SVM enforce the boundary between guest ring 0 and host ring 0 in silicon; a guest running in VMX non-root mode cannot issue a host syscall directly. But that boundary has bugs. KVM has had privilege escalation CVEs, and the entire history of hypervisor security suggests that "assume the hardware boundary is unbreakable" is not a viable threat model at scale.

The Jailer (`firecracker-jailer`) is a separate binary that wraps the Firecracker process: it drops privileges, sets up a PID namespace, configures cgroups, and calls `pivot_root` before exec-ing `firecracker`. What the Jailer does not do is install any seccomp filter on the Firecracker process. Seccomp installation is Firecracker's own responsibility, performed per-thread from within the process.

Seccomp is the second containment layer, sitting between the hardware boundary and the Jailer's namespace isolation:



None of the three layers depends on the others being intact. If KVM's boundary holds, the guest never reaches seccomp. If KVM's boundary is breached, seccomp limits the thread to its allowlist. If both are breached, the Jailer's cgroup limits, restricted namespace, and dropped UID constrain what the process can reach on the host. Each layer independently limits the blast radius of a compromise at its own level.

Concretely: a vCPU thread that has been taken over by a guest escape and is now running attacker code on the host cannot open a socket, cannot fork, cannot exec a new binary, cannot load a kernel module, cannot mount a filesystem, and cannot issue a KVM VM-level ioctl that only the VMM thread is permitted to issue. What it can do is call `KVM_RUN` again — which puts it back in the guest.

The argument-level `ioctl` filter is the sharpest edge of this. Without it, a vCPU thread that could call `ioctl` with an arbitrary request argument could issue `KVM_CREATE_VM` (a VM-level ioctl), `KVM_SET_USER_MEMORY_REGION` (which maps host memory into the guest), or `TIOCSTI`. With it, the only ioctls available are the 19 vCPU-specific operations in the allowlist.

Two Distinct Seccomp Layers

Firecracker's seccomp filters are host-side: they restrict syscalls made by the Firecracker process against the host kernel. When Firecracker hosts containers via `firecracker-containerd`, a second independent seccomp layer operates inside the guest:

Layer	Filters	Installed by	Evaluated by
Firecracker host seccomp	Host syscalls from the Firecracker process	Firecracker itself	Host kernel
OCI/runc seccomp	Guest syscalls from container workloads	`runc` inside the guest	Guest kernel

Both layers use the identical kernel mechanism: `seccomp(SECCOMP_SET_MODE_FILTER, flags, &sock_fprog)` on x86-64. The OCI runtime spec's seccomp field maps `SCMP_ACT_*` action names and `SCMP_CMP_*` operators to the same kernel constants described in this chapter; the underlying BPF evaluation is identical. The two layers do not interact: the host kernel evaluates the Firecracker filter, and the guest kernel evaluates the container filter. A call that passes the guest filter but would fail the host filter never reaches the host — the guest kernel handles it without issuing a hypercall.

The containerd book covers the OCI/runc seccomp path in detail.

Custom Filters and the `--no-seccomp` Flag

> ***Note:*** *`--no-seccomp` disables all BPF filtering. Firecracker's documentation states explicitly: "Do not use in production." A Firecracker process running without seccomp has no second containment layer and relies solely on KVM hardware isolation. Running with `--no-seccomp` on a host that serves untrusted workloads removes a significant portion of the defense-in-depth design.*

Two CLI flags control filter selection:

- `--no-seccomp`: passes an empty `bpf_filter` slice to `apply_filter`, which returns immediately at step 1 without installing any filter.
- `--seccomp-filter <path>`: loads a pre-compiled bitcode-serialized `.bpf` file, overriding the embedded defaults. This file is produced by `seccompiler-bin --target-arch x86_64 --input-file policy.json --output-file out.bpf`. Custom filters are useful for experimental GNU libc targets (which ship without embedded default filters), debug builds, and rapid production mitigation without recompiling Firecracker.

`seccompiler-bin` also accepts `--split-output` (writes one `<thread>.bpf` file per thread, used in the test suite) and `--target-arch (x86_64 | aarch64)`.

A Brief History

Firecracker's seccomp policy has changed only a handful of times since the initial open-source release:

Version	Change
v0.13.0	Changed the default from seccomp-level 0 (no filtering) to level 2 (full argument-level filtering) for all threads.
v0.15.1	Added `madvise` to the VMM allowlist after musl's allocator called it under memory pressure, causing random VM terminations.
v1.11.0	Migrated the compiler backend from hand-written Rust BPF code generation to libseccomp (PR #4926, merged 2025-01-16); added the empty BPF slice path for debug builds.
PR #5298	Replaced `HashMap` with `BTreeMap` in the compilation pipeline, fixing a reproducibility bug where identical inputs produced different BPF bytecode across builds.

The v0.15.1 `madvise` incident is a useful calibration. A tight seccomp allowlist is not a set-and-forget policy; it is a contract between the policy and the implementation. When the implementation changes — a new dependency, a new libc version, a new code path under memory pressure — the policy must change with it. The cost of getting this wrong is an unexplained VM termination, not a helpful error message. That is a steep debugging tax, and it is the price of running with a minimal allowlist.

Sources And Further Reading

- Firecracker seccomp documentation: https://github.com/firecracker-microvm/firecracker/blob/main/docs/seccomp.md
- Firecracker seccompiler documentation: https://github.com/firecracker-microvm/firecracker/blob/main/docs/seccompiler.md
- x86-64 default policy file: https://github.com/firecracker-microvm/firecracker/blob/main/resources/seccomp/x86_64-unknown-linux-musl.json
- Firecracker `apply_filter` implementation: https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/seccomp.rs
- Firecracker seccompiler library: https://github.com/firecracker-microvm/firecracker/blob/main/src/seccompiler/src/lib.rs
- `SeccompCondition::to_scmp_type()` (dword/qword masking): https://github.com/firecracker-microvm/firecracker/blob/main/src/seccompiler/src/types.rs
- Firecracker design document: https://github.com/firecracker-microvm/firecracker/blob/main/docs/design.md
- Jailer documentation: https://github.com/firecracker-microvm/firecracker/blob/main/docs/jailer.md

- PR #4926 — libseccomp backend migration: https://github.com/firecracker-microvm/firecracker/pull/4926
- PR #5298 — BTreeMap for reproducible BPF output: https://github.com/firecracker-microvm/firecracker/pull/5298
- `seccomp(2)` man page: https://man7.org/linux/man-pages/man2/seccomp.2.html
- Kernel seccomp filter documentation: https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html
- `include/uapi/linux/seccomp.h` — action constants: https://github.com/torvalds/linux/blob/master/include/uapi/linux/seccomp.h
- x86-64 syscall table: https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall_64.tbl
- OCI Runtime Spec, Linux seccomp field: https://github.com/opencontainers/runtime-spec/blob/main/config-linux.md
- `firecracker-containerd`: https://github.com/firecracker-microvm/firecracker-containerd
- `rust-vmm/seccompiler` crate (v0.5.0, released 2025-03-07): https://github.com/rust-vmm/seccompiler/blob/main/src/lib.rs

Chapter 20: The Threat Model

If you run untrusted code on a multi-tenant system, the central question is not whether isolation works in the happy path — it does. The question is what an adversary can reach after breaking the first barrier. Containers answer it one way: a process that escapes its namespace lands in the host kernel, which is also the kernel every other tenant is running on. MicroVMs answer it differently, with a sequence of barriers, each designed to contain what the previous one failed to stop.

This chapter maps that sequence in Firecracker's terms. The barriers are not informal defense-in-depth platitudes. They are hardware CPU modes, per-thread BPF filters, and a setuid jail binary — each with a specific syscall, VMCS field, or kernel module parameter attached to the claim.

The Trust Axiom

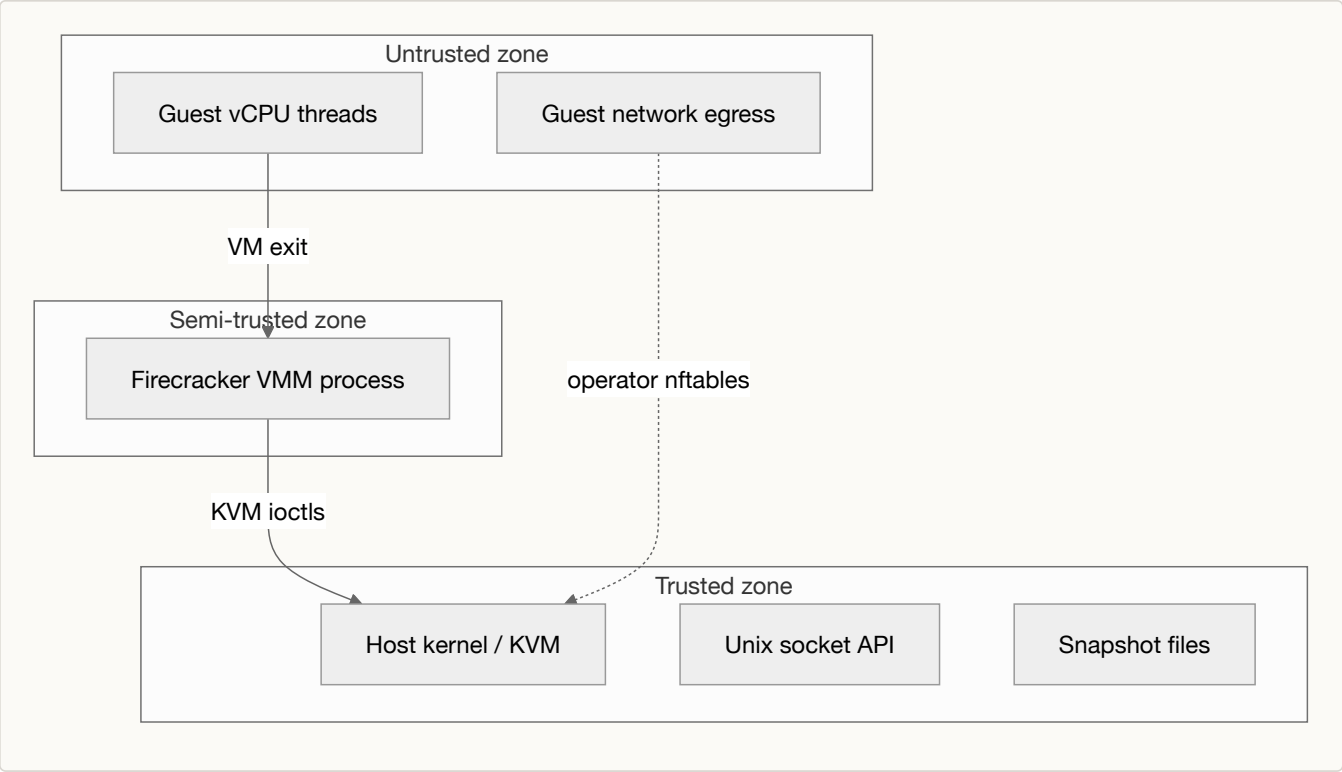
Firecracker's design document states the baseline assumption plainly:

> *"all vCPU threads are considered to be running malicious code as soon as they have been started; these malicious threads need to be contained."*

This is not a disclaimer hedging an improbable edge case. It is the load-bearing premise the entire architecture rests on. A guest OS that boots correctly and runs cooperative workloads is operationally convenient; the containment model is designed for a guest that has been taken over and is actively probing for weaknesses in every direction.

From that premise, the trust hierarchy partitions into three zones: an untrusted zone (all guest vCPU threads and all guest network traffic), a semi-trusted zone (the Firecracker VMM process itself), and a trusted zone (the host kernel, KVM module, the Unix socket API channel, snapshot files, and the physical hardware). "Semi-trusted" is precise: Firecracker is written in safe Rust with a deliberately small codebase, but it remains a userspace process, and a vulnerability in its virtio emulation paths is in scope for exploitation. What limits the damage is the containment imposed around that process — not the assumption that the process is correct.

Firecracker explicitly disclaims network traffic filtering. The design document states that all egress from a guest is untrusted and must be filtered at the host level by the operator — typically with `nftables` rules applied to the TAP interfaces. That delegation is not a weakness; it reflects where the right tool sits. The VMM is not a firewall.



Layer 1: Hardware Virtualization

The outermost barrier is the one the CPU enforces without software assistance. Intel VMX introduces two orthogonal operating modes: VMX root operation, where the hypervisor and host OS run, and VMX non-root operation, where the guest runs. Each mode retains the usual CPL 0--3 ring hierarchy, but a guest OS running at CPL 0 in VMX non-root mode does not have full ring-0 privilege. Every privileged action the guest attempts — writing `CR3`, accessing an MSR, executing `INVLPG` — is governed by the VM-execution control fields in the VMCS (Virtual Machine Control Structure), and most of them cause a VM exit rather than executing.

On a VM exit, the CPU atomically loads host state from the VMCS host-state area — `CR0`, `CR3`, `CR4`, segment selectors, `RIP` and `RSP` from the `HOST_RIP` and `HOST_RSP` fields — and saves guest state. Linux's `arch/x86/kvm/vmx/vmenter.S` notes this directly: "After a successful `VMRESUME`/`VMLAUNCH`, control flow 'magically' resumes below at `vmx_vmexit` due to the VMCS `HOST_RIP` setting." The guest did not transfer control voluntarily; the CPU forced the transition and simultaneously switched to a separate address space.

That `vmenter.S` path also zeroes all general-purpose registers except RSP and RBX before returning to host code, preventing speculative use of guest register values in host execution paths. RSB (Return Stack Buffer) clearing and `SPEC_CTRL` MSR handling are applied as post-exit mitigations for Spectre-class side channels — more on those below.

AMD SVM is structurally parallel. The VMCB (Virtual Machine Control Block) is divided into a control area (intercepts, ASID, ASID flush bits) and a save area (guest register state). `VMRUN` saves host state to the area pointed at by the `HSAVE_PA` MSR and enters the guest; `#VMEXIT` reverses it.

The KVM API exposes this hardware boundary through the three-scope ioctl hierarchy. Applications verify `KVM_GET_API_VERSION` returns `12`, create a VM with `KVM_CREATE_VM` (`_IO(KVMIO, 0x01)`) on `/dev/kvm`, register guest memory with `KVM_SET_USER_MEMORY_REGION` using `struct kvm_userspace_memory_region` (slot, flags, `guest_phys_addr`, `memory_size`, `userspace_addr`), and run a vCPU with `KVM_RUN` (`_IO(KVMIO, 0x80)`, decimal `44672`). When a guest action requires VMM intervention, KVM sets `kvm_run->exit_reason` in the shared mmap region and returns. Common exit reasons include `KVM_EXIT_IO` (2) for port I/O, `KVM_EXIT_MMIO` (6) for MMIO, and `KVM_EXIT_SHUTDOWN` (8) for guest shutdown. Operations the host kernel can handle entirely in KVM — local APIC, IOAPIC, PIT — never cross the `KVM_RUN` boundary to userspace at all.

The guarantee this layer provides is direct: guest code cannot read or write host memory, cannot execute privileged host instructions, and cannot modify host page tables. What it does not guarantee is that KVM's own kernel-mode code is bug-free — and that caveat is exactly where CVE-2021-29657 sits.

Layer 2: Seccomp BPF Filters

Chapter 19 walked through the `seccomp(2)` mechanism and Firecracker's filter allow-lists in detail. Here the relevant frame is what the filters contribute to the layered barrier.

Suppose a guest has found a bug in the KVM hardware boundary and is now executing arbitrary code inside a vCPU thread on the host. Without any further containment, that thread can call every syscall the process is permitted to call: `socket`, `execve`, `fork`, `mount`. The host kernel evaluates each one against the Firecracker process's credentials and the host's network configuration. A guest that can issue arbitrary syscalls on the host has escaped.

Seccomp BPF filters answer this by restricting what syscalls each thread in the Firecracker process can reach, before examining whether any individual call is malicious. The filter policy is not system-wide; it is per-thread and applied from three distinct JSON sources compiled at build time by `seccompiler-bin` into BPF bytecode embedded in the `firecracker` binary. The three thread categories and their approximate allow-list sizes on `x86_64-unknown-linux-musl` (main branch):

- The `vmm` thread allows roughly 68 distinct syscalls, covering virtio I/O, KVM VM-scope ioctls (`KVM_SET_USER_MEMORY_REGION`, `KVM_IOEVENTFD`, `KVM_IRQFD`, `KVM_GET_DIRTY_LOG`), and TUN/TAP ioctls (`TUNSETIFF`, `TUNSETOFFLOAD`, `TUNSETVNETHDRSZ`).
- The `api` thread allows roughly 41 syscalls and zero KVM ioctls — only `FIONBIO` (value `21537`) for non-blocking I/O control, plus the Unix socket calls it actually needs.
- The `vcpu` thread allows roughly 47 syscalls, principally the vCPU-scope ioctls including `KVM_RUN` (44672), but nothing beyond what vCPU execution requires.

The `default_action` across all three filters is `"trap"` — mapping to `SECCOMP_RET_TRAP`, which delivers `SIGSYS`. An unlisted call does not return an error; it terminates the thread. An operator can supply a custom pre-compiled filter via `--seccomp-filter`, but the default posture is deny-by-default.

Beyond the syscall number, Firecracker uses the argument-evaluation capability of seccomp BPF — the filter receives `struct seccomp_data.args[6]` and can inspect up to six arguments, though it cannot dereference pointers. A handful of constraints that matter for an escape scenario:

- `mmap` requires that `PROT_EXEC` (bit 2) be unset, so a compromised VMM process cannot create a new executable mapping to JIT shellcode.
- `mprotect` carries the same `PROT_EXEC` exclusion, preventing an attacker from making an existing mapping executable.
- `socket` permits only `AF_UNIX` (value `1`). No `AF_INET` or `AF_INET6` sockets are reachable from any thread in the Firecracker process.
- `tkill` is restricted to signals `6` (SIGABRT) and `35` (SIGRTMIN + the per-vCPU RT signal offset), blocking arbitrary signal delivery to host threads.

The combined effect: a compromised VMM process cannot open a network socket, cannot JIT executable code, and cannot escalate through signal tricks. It is constrained to the exact ioctls and syscalls Firecracker itself needs to run.

Installing a seccomp BPF filter requires either `CAP_SYS_ADMIN` or a prior `prctl(PR_SET_NO_NEW_PRIVS, 1)` call; the kernel returns `-EACCES` otherwise. Firecracker uses `PR_SET_NO_NEW_PRIVS` — which also prevents `execve` from granting the child more privileges than the parent, closing an escalation path before any filter is in place.

Layer 3: The Jailer

The jailer is a separate setuid binary. Its job is to set up every privileged resource Firecracker needs, then `exec()` into the `firecracker` binary. After that handoff, `firecracker` can only access resources the jailer explicitly created inside the jail before transferring control.

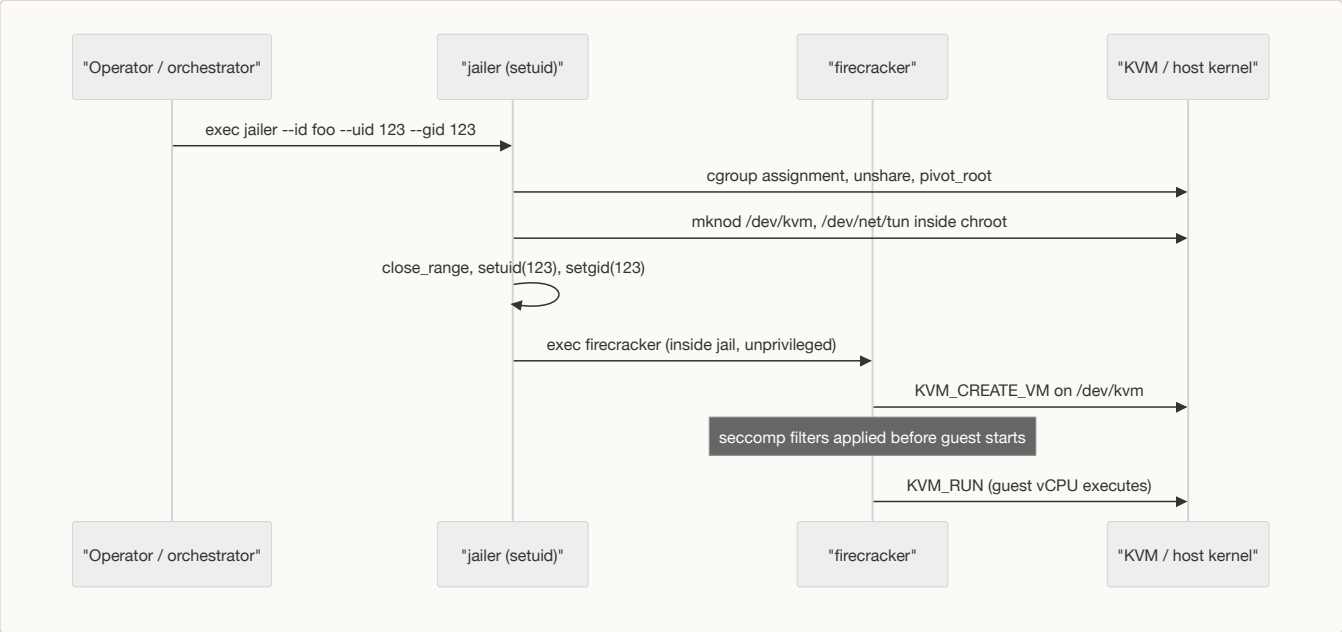
The sequence of operations the `jailer` binary performs, in order:

1. Places the process into a cgroup (v1: by writing to the `tasks` file under one of `cpuset`, `cpu,cpuacct`, `memory`, `net_cls,net_prio`, or `pids`; v2: by writing to `cgroup.procs`). The `--cgroup-version` flag selects which (default: v1).
2. Creates a new mount namespace with `unshare()`, then calls `pivot_root()` — not the older `chroot()` — to establish a jail root at `<chroot_base>/<exec_file_name>/<id>/root`.
3. Creates only the device nodes Firecracker actually needs inside the chroot: `/dev/net/tun` and `/dev/kvm`. Nothing else is `mknod()`'d.
4. Optionally creates a PID namespace via `clone(CLONE_NEWPID)` (the `--new-pid-ns` flag) and a network namespace via `setns(fd, CLONE_NEWNET)` (the `--netns` flag).

5. Closes all file descriptors that were not explicitly inherited, using `close_range(3, UINT_MAX, CLOSE_RANGE_UNSHARE)` (`close_range` syscall, requires kernel 5.9+).
6. Drops privilege with `setuid(uid)` and `setgid(gid)` to a unique non-privileged uid/gid per instance.
7. Sets resource limits: `setrlimit(RLIMIT_FSIZE)` and `setrlimit(RLIMIT_NOFILE)`, the latter defaulting to 2048 file descriptors.
8. `exec()`s into `firecracker`.

The use of `pivot_root()` rather than `chroot()` is meaningful. `chroot()` only changes the root directory for path resolution; a process with `CAP_SYS_CHROOT` can break out of a `chroot()` jail. `pivot_root()` replaces the entire mount namespace root, so the old filesystem tree is genuinely unreachable unless the jailer explicitly binds it in — which it does not.

The jailer's own inputs are treated as trusted. The jailer documentation is explicit: it is the operator's responsibility to ensure that jailer input paths cannot be tampered with by unprivileged local users. The jailer does not defend against a malicious operator; it defends against a compromised `firecracker` process trying to reach host resources.



Device Model as Attack Surface Reduction

The device model bounds what the attacker has to aim at in the first place — not by filtering, but by the code not being present.

Firecracker's emulated device set is: VirtIO Net, VirtIO Block, and VirtIO Vsock (all over a virtio-mmio transport with I/O rate limiting); a serial console (8250 UART); a partial i8042 keyboard controller used only for reboot signaling; and the PIC, IOAPIC, and PIT handled entirely within KVM's in-kernel emulation. That is the complete list. There is no PCI bus, no BIOS or firmware, no ACPI, no USB

controller, no GPU, no floppy disk controller, no sound device, no legacy ISA devices beyond i8042 and 8250. The guest boots via a direct-boot protocol straight to the Linux kernel, with no firmware layer in the path.

This absence is a security property. VENOM (CVE-2015-3456) exploited the floppy disk controller emulation in QEMU: `fdctrl_handle_drive_specification_command()` allocated a 512-byte FIFO buffer, but a missing `data_pos` reset in one branch of the fifth-parameter handling allowed the write pointer to advance past the buffer boundary on every subsequent I/O byte. A privileged guest user writing to the FDC I/O port could overflow the heap region immediately following that FIFO. CVSS 2 score: 7.7 HIGH, fixed via commit `e907746266721f305d67bc0718795fedee2e824c`, released in QEMU after 2.3.0. In Firecracker, the code path does not exist. You cannot exploit emulation for a device the VMM did not implement.

The same logic applies to the virtio descriptor table, which Firecracker does parse. The virtio 1.2 specification defines `struct virtq_desc` as 16 bytes: `le64 addr` (guest-physical), `le32 len`, `le16 flags`, `le16 next`. The flags field carries `VIRTQ_DESC_F_NEXT` (bit 0, value 1) for descriptor chaining and `VIRTQ_DESC_F_WRITE` (bit 1, value 2) for write-only device buffers. Maximum queue size is 32,768 entries. The spec places a MUST-level obligation on the VMM at section 2.7.5.1: "A device MUST NOT write to any descriptor table entry." Equally, the VMM must validate all descriptor fields before acting on them — the `addr`, `len`, and `next` fields are all guest-controlled and all could be crafted to manipulate host memory if bounds checks are missing.

CVE-2019-14835 is the canonical example of what happens when that validation is absent. The `get_indirect()` function in the vhost-net kernel driver (`drivers/vhost/vhost.c`) iterated up to `USHRT_MAX + 1` (65,536) times writing to a log buffer during live migration, without checking that `*log_num` stayed within the actual buffer size. A guest could craft descriptor tables with large `len` values to trigger the overflow during a migration event, achieving kernel heap overflow. CVSS 3.1: 7.8 HIGH. Introduced in Linux 2.6.34, fixed in Linux 5.3, commit `060423bfdee3f8bc6e2c1bac97de24d5415e2bc4`. This was a kernel-mode virtio backend, not a userspace VMM, but the attack surface is the same: guest-controlled descriptor table content reaching code that fails to validate it.

Comparison to Container Isolation

The contrast between container isolation and microVM isolation is not a matter of degree. It is a categorical difference in what the attacker reaches if the first barrier breaks.

A container shares the host kernel. Every syscall a containerized process issues goes directly to the same kernel all other containers on the host are running on. Docker's default seccomp profile blocks approximately 44 syscalls out of 300-plus, leaving roughly 256 reachable to the host kernel by default. Research from 2025 (arxiv:2510.03720) showed that optimized per-container syscall limiting can reduce

the average allowed set to roughly 87 syscalls, which would statically prevent exploitation of 87 CVEs in the study's dataset — a meaningful improvement, but still operating entirely within the shared-kernel model.

Consider CVE-2022-0847, Dirty Pipe. The commit `f6dd975583bd` ("pipe: merge anon_pipe_buf*_ops"), introduced in Linux 5.8, left the `PIPE_BUF_FLAG_CAN_MERGE` flag in `struct pipe_buffer` uninitialized. An unprivileged process could fill all pipe ring slots — setting the flag on each — then `splice()` a read-only file's page into the pipe (inheriting the flag from the ring), then `write()` to append into the page cache, silently overwriting read-only file content without write permission. Fixed in Linux 5.16.11, 5.15.25, and 5.10.102 on 2022-02-23.

From a container, this attack calls `pipe(2)`, `splice(2)`, and `write(2)` — all syscalls Docker's default profile permits — directly into the host kernel's syscall handler. The result is host page cache overwrite. From a microVM guest, those same calls go to the guest OS kernel. The host kernel does not see them. Reaching the host would require first escaping the hardware VMX/SVM boundary, surviving the seccomp filter, and escaping the jailer's `pivot_root()` jail — three barriers that are not present in the container model.

Similarly, CVE-2017-7308 let an unprivileged user reach privilege escalation by crafting `setsockopt()` calls via `AF_PACKET` sockets. From a container, `socket(AF_PACKET, ...)` may reach the host kernel depending on the container's seccomp profile. From a Firecracker VMM process, `socket()` is allowed by the seccomp filter only for `AF_UNIX` (value `1`) — `AF_PACKET` is not on the list, and the default action is `SECCOMP_RET_TRAP`.

Property	Container (default Docker)	Firecracker microVM
Kernel boundary	Namespace + cgroup (shared kernel)	Hardware VMX/SVM + KVM
Syscall surface to host	~256 of 300+ reachable	vcpu thread: ~47 via seccomp
Default seccomp posture	~44 syscalls blocked	All threads: `default_action=trap`
Device attack surface	Full host driver stack	8 emulated devices; no PCI/BIOS/USB
Escape path	Single shared-kernel bug sufficient	KVM escape + VMM exploit + seccomp bypass + jailer escape

The KVM Boundary Is Not Inviolable

The hardware virtualization boundary stops the vast majority of guest-originating attacks because it is enforced by CPUs that AMD and Intel have spent decades hardening. But the boundary is not a proof — it is an engineering artifact, and KVM's kernel-mode code is in scope for bugs.

CVE-2021-29657 was the first public writeup of a KVM guest-to-host breakout that did not rely on any bug in QEMU or a userspace VMM at all. Affected kernels: v5.10 through v5.12-rc6, patched in March 2021. The attack targeted KVM's AMD SVM-specific kernel-mode code directly from a guest vCPU, without requiring the guest to first manipulate the VMM process; Intel VMX users were not affected. A comparable Intel-VMX-specific guest-to-host breakout has not been publicly demonstrated. The CVE is nonetheless a proof of concept that the "KVM escape" scenario the rest of the containment model is designed for is not purely theoretical.

The practical implication for the threat model is this: the seccomp filters and the jailer are not fallback measures deployed on the assumption that the hardware boundary works. They are independent containment layers designed for the scenario where the hardware boundary has already failed.

Microarchitectural Side Channels

Software barriers are not the only threat surface. Several CPU microarchitectural vulnerabilities allow cross-tenant information leakage through shared hardware state that the VMM cannot observe or block in software.

CVE-2018-3646 (L1TF / Foreshadow-VMM) is the defining example. An x86 PTE with the Present bit cleared causes speculative execution to load the physical address from the L1D cache before the page fault is raised and before the permission check that would stop it. With Hyper-Threading enabled, a guest vCPU running on one logical processor can speculatively read L1D contents populated by the host on the sibling logical processor of the same physical core. The mitigation MSR is `IA32_FLUSH_CMD` at address `0x10B` — write-only; writing bit 0 (`L1D_FLUSH` , value `1`) flushes and invalidates the L1D on the executing physical core. Support is enumerated via `CPUID.(EAX=07H,ECX=0):EDX[28]` . Susceptibility can be checked via `IA32_ARCH_CAPABILITIES` MSR at `0x10A` ; bit 0 (`RDCL_NO` , value `1`) indicates the processor is not vulnerable. KVM exposes the mitigation via `/sys/module/kvm_intel/parameters/vmentry_l1d_flush` : `cond` (flush only after non-audited code paths, default) or `always` (unconditional, with 1--50% performance overhead depending on VM exit rate).

CVE-2017-5715 (Spectre v2) targets the branch predictor. The mitigation MSRs are `IA32_SPEC_CTRL` at `0x48` (IBRS = bit 0, STIBP = bit 1) and `IA32_PRED_CMD` at `0x49` (IBPB = bit 0, write-only). KVM exposes these to guests and handles the host-side save/restore: on VM exit, for CPUs using classic IBRS, KVM sets IBRS to 0; CPUs with enhanced IBRS (eIBRS, widely available since ~2019) do not require this

per-exit write because eIBRS protection persists across the transition. On VM entry KVM restores the guest's saved IBRS value. The RSB is flushed on every VM exit. Current mitigation status is visible at `/sys/devices/system/cpu/vulnerabilities/spectre_v2`.

Neither of these is a VMM bug in the conventional sense. They are properties of the physical hardware, and the software mitigation for both converges on the same recommendation in Firecracker's production host setup guide: disable SMT (Hyper-Threading) entirely. With SMT disabled, no sibling logical processor can speculatively read L1D contents belonging to another tenant. The L1D flush MSR then becomes a belt-and-suspenders measure rather than the primary defense.

A complete production host mitigation table, drawn from Firecracker's `prod-host-setup.md`:

Mitigation	Mechanism	Threat addressed
Disable SMT	Kernel boot parameter or BIOS	Cross-tenant L1D leakage via sibling threads (L1TF)
Disable KSM	`echo 0 > /sys/kernel/mm/ksm/run`	Cross-tenant timing attacks via page deduplication
Disable swap	`swapoff -a`	Guest memory remanence on storage media
DDR4 with TRR + ECC	Hardware selection	Rowhammer
`kvm.nx_huge_pages=never`	Kernel module parameter	iTLB multihit regression (Linux 6.1, x86-64)
Updated CPU microcode	Distribution security updates	All speculative execution side channels

> ***Note:*** *Disabling KSM (`/sys/kernel/mm/ksm/run`) and swap require root on the host. Consult your platform's hardening guide before making these changes in production.*

On ARM, Firecracker resets the `CNTPCT` physical counter only when `KVM_CAP_COUNTER_OFFSET` is available, which requires kernel 6.4 or later.

Snapshot Trust and Operational Hazards

Snapshot files — the VM state file, the memory snapshot, and the disk image — are classified as trusted in Firecracker's threat model. This is not a strong guarantee. Firecracker applies a 64-bit CRC to the VM state file for partial corruption detection; it does not authenticate or encrypt snapshot content, and the CRC covers only the state file, not the memory snapshot or the disk image. All three files must be independently secured by the operator — an attacker who can modify a snapshot file can inject arbitrary guest state.

Resuming identical snapshots multiple times creates a subtler hazard: UUID collisions, reuse of entropy pool state, repeated cryptographic tokens, and reused RNG seeds across multiple resumed instances. If snapshot triggering is exposed to customers, operators must enforce disk quotas to prevent DoS via unbounded snapshot files.

There are two configuration hazards worth naming before a deployment reaches production. The 8250 serial device can cause unbounded memory and storage usage on the host if guest output is not rate-limited; the production guidance is to disable it with the kernel command line argument `8250.nr_uarts=0`. The MMDS (MicroVM Metadata Service) is accessible from the guest at `169.254.169.254` by default; operators must block it at the host with an `nftables` rule targeting TAP interfaces:

> ***Note:*** *The commands below modify the host firewall. The `firecracker` table and `filter` chain must exist before adding the rule; create them once if they do not (see `prod-host-setup.md` for the full setup). Verify the rule does not conflict with existing `nftables` rulesets before applying.*

```
nft add table ip firecracker
nft add chain ip firecracker filter { type filter hook forward priority 0 \; }
nft add rule ip firecracker filter iifname "tap*" ip daddr 169.254.169.254 counter drop
```

The threat model fixes the adversary's position and the defender's posture; Chapter 21 covers how to validate both in practice, using automated policy checks and runtime attestation to confirm that the barriers described here are actually in place on a production host.

Sources And Further Reading

- Firecracker design document: https://github.com/firecracker-microvm/firecracker/blob/main/docs/design.md
- Firecracker jailer documentation: https://github.com/firecracker-microvm/firecracker/blob/main/docs/jailer.md
- Firecracker production host setup guide: https://github.com/firecracker-microvm/firecracker/blob/main/docs/prod-host-setup.md
- Firecracker seccomp documentation: https://github.com/firecracker-microvm/firecracker/blob/main/docs/seccomp.md
- Firecracker seccompiler documentation: https://github.com/firecracker-microvm/firecracker/blob/main/docs/seccompiler.md
- Firecracker snapshot support documentation: https://github.com/firecracker-microvm/firecracker/blob/main/docs/snapshotting/snapshot-support.md
- Firecracker x86-64 seccomp filter JSON: https://github.com/firecracker-microvm/firecracker/blob/main/resources/seccomp/x86_64-unknown-linux-musl.json
- KVM API documentation: https://www.kernel.org/doc/html/latest/virt/kvm/api.html

- Linux kernel KVM UAPI header (`kvm.h`): https://github.com/torvalds/linux/blob/master/include/uapi/linux/kvm.h
- Linux kernel `vmenter.S`: https://github.com/torvalds/linux/blob/master/arch/x86/kvm/vmx/vmenter.S
- Kernel seccomp_filter documentation: https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html
- L1TF (CVE-2018-3646) kernel documentation: https://docs.kernel.org/admin-guide/hw-vuln/l1tf.html
- Spectre v2 (CVE-2017-5715) kernel documentation: https://docs.kernel.org/admin-guide/hw-vuln/spectre.html
- OASIS virtio v1.2 specification: https://docs.oasis-open.org/virtio/virtio/v1.2/virtio-v1.2.html
- Intel SDM Vol. 3C (VMX architecture): https://cdrdv2-public.intel.com/789585/326019-sdm-vol-3c.pdf
- NVD entry for CVE-2015-3456 (VENOM): https://nvd.nist.gov/vuln/detail/CVE-2015-3456
- CrowdStrike VENOM technical disclosure: https://www.crowdstrike.com/en-us/blog/venom-vulnerability-details/
- NVD entry for CVE-2019-14835 (vhost-net): https://nvd.nist.gov/vuln/detail/CVE-2019-14835
- oss-security disclosure for CVE-2019-14835: https://www.openwall.com/lists/oss-security/2019/09/17/1
- LWN case study on CVE-2021-29657 (KVM guest-to-host breakout): https://lwn.net/Articles/861330/
- Dirty Pipe (CVE-2022-0847) canonical writeup: https://dirtypipe.cm4all.com/
- NVD entry for CVE-2022-0847: https://nvd.nist.gov/vuln/detail/cve-2022-0847
- Docker seccomp documentation: https://docs.docker.com/engine/security/seccomp/
- Syscall limitation research (arxiv:2510.03720): https://arxiv.org/html/2510.03720v1

PART VI — INTEGRATION AND ECOSYSTEM

Chapter 21: Host Networking For MicroVMs

When Firecracker boots a guest kernel, the guest is completely isolated from the host network: no shared namespace, no injected interface, no route added by the runtime. The question is how traffic gets in and out. A container reaches its bridge through a veth pair the runtime handed to a CNI plugin; a microVM has its own kernel, so there is nothing to share — every byte that crosses the isolation boundary must pass through a file descriptor the VMM process owns. That file descriptor is the TAP device, and it is the whole story.

The TAP Device and the VMM File Descriptor

A **TAP** device (network TAP) is a kernel virtual Ethernet interface whose packet stream is exposed to a userspace process through a file descriptor on `/dev/net/tun` (character device major 10, minor 200). The kernel document at `Documentation/networking/tuntap.rst` describes two variants: **TUN** at layer 3 (raw IP datagrams) and **TAP** at layer 2 (Ethernet frames including MAC header). Firecracker uses TAP because virtio-net presents Ethernet semantics to the guest: the guest's driver sees a virtual NIC with a MAC address, sends and receives full Ethernet frames, and never knows those frames are travelling through a file descriptor in the VMM process on the other side.

Opening the device requires `CAP_NET_ADMIN`. Firecracker opens `/dev/net/tun` with:

```
libc::open(c"/dev/net/tun".as_ptr(), libc::O_RDWR | libc::O_NONBLOCK | libc::O_CLOEXEC)
```

Each flag matters. `O_RDWR` gives bidirectional packet access — the same fd both dequeues incoming frames and injects outgoing ones. `O_NONBLOCK` makes `read` and `write` return `EAGAIN` rather than blocking, which is essential for Firecracker's epoll-driven event loop: the I/O thread can poll multiple event sources without getting stuck in a slow system call. `O_CLOEXEC` ensures the fd does not leak into child processes; Firecracker's `jailer` forks before execing the VMM, and a leaked TAP fd in the jailer's pid namespace would outlive the microVM.

After `open`, Firecracker calls `ioctl(fd, TUNSETIFF, &ifreq)`. `TUNSETIFF` is defined in `include/uapi/linux/if_tun.h` as `_IOW('T', 202, int)` — type `'T'` (decimal 84), sequence 202. The `ifreq.ifr_name` field sets the interface name (maximum `IFNAMSIZ` = 16 bytes including the NUL terminator); an empty name lets the kernel assign the next available `tapN`. Firecracker's generated bindings (`src/vmm/src/devices/virtio/net/generated/if_tun.rs`) set three flags in `ifreq.ifr_flags`:

Flag	Value	Effect
`IFF_TAP`	2	Layer-2 (Ethernet) mode
`IFF_NO_PI`	4096	Suppress the 4-byte `struct tun_pi` protocol info header
`IFF_VNET_HDR`	16384	Prepend/consume a `virtio_net_hdr` on each frame

`IFF_NO_PI` removes a 4-byte prefix the kernel otherwise adds to every read (it carries the EtherType and flags that are already in the Ethernet header). `IFF_VNET_HDR` is the productive one: it tells the kernel to expect a `virtio_net_hdr` structure at the front of every frame written through the fd, and to prepend one to every frame read from it. That header carries checksum-offload and segmentation-offload metadata between the guest driver and the host NIC, allowing large sends to pass through the VMM without being fragmented in the VMM's own memory.

Two more ioctls follow during device activation. `TUNSETOFFLOAD` (`_IOW('T', 208, unsigned int)`) accepts a `TUN_F_*` bitmask derived from the virtio features the guest negotiated: `TUN_F_CSUM` (0x01), `TUN_F_TSO4` (0x02), `TUN_F_TSO6` (0x04), `TUN_F_TSO_ECN` (0x08), and `TUN_F_UFO` (0x10). `TUNSETVNETHDRSZ` (`_IOW('T', 216, int)`) is called with 12, the size of `virtio_net_hdr_v1`, telling the kernel exactly how many bytes of virtio header to expect on each transfer.

Firecracker never calls `TUNSETPERSIST`. The TAP device therefore lives only as long as the fd is open — when the `Tap` struct drops, the fd closes, and the kernel removes the interface. Device lifetime is bound to the VMM process with no cleanup step required.

The VMM's Use of the TAP fd in the Event Loop

The `Tap` struct in `src/vmm/src/devices/virtio/net/tap.rs` wraps a single `File` that owns the fd. At runtime, the net device registers five event sources with Firecracker's epoll manager:

Token	Source	Purpose
`PROCESS_VIRTQ_RX`	RX virtqueue eventfd	Guest driver has posted receive buffers
`PROCESS_VIRTQ_TX`	TX virtqueue eventfd	Guest driver has posted transmit buffers
`PROCESS_TAP_RX`	TAP fd (EPOLLIN, edge-triggered)	Frame has arrived from the host
`PROCESS_RX_RATE_LIMITER`	RateLimiter timerfd	RX token bucket has refilled
`PROCESS_TX_RATE_LIMITER`	RateLimiter timerfd	TX token bucket has refilled

`EPOLLIN` on the TAP fd triggers `process_rx()` — the handler reads a frame from the host network stack and writes it into the guest's receive ring. The other direction is driven not by the TAP fd but by an eventfd: when the guest driver writes the virtio MMIO notify register, KVM translates that write into an eventfd notification (`PROCESS_VIRTQ_TX`), and the VMM dequeues the transmit buffer, runs it through the rate limiter, and calls `libc::writev()` on the TAP fd to inject the Ethernet frame into the kernel.

This asymmetry is worth pausing on. The guest signals the VMM through a KVM eventfd bound to the MMIO address (the standard virtio/KVM notification path); the host signals the VMM through the TAP fd being readable. Two distinct kernel mechanisms, unified by the same epoll loop.

virtio-net Queues and the vnet Header

Firecracker's virtio-net device (`VIRTIO_ID_NET = 1`) uses exactly two virtqueues (`NET_NUM_QUEUES = 2`), each of depth 256 (`NET_QUEUE_MAX_SIZE = 256`). Queue 0 (`RX_INDEX`) holds device-writable buffers the guest driver posts for incoming frames. Queue 1 (`TX_INDEX`) holds device-readable buffers the guest driver fills with outgoing frames. The maximum per-buffer size is 65,562 bytes: a 12-byte vnet header plus the largest Ethernet frame.

With `IFF_VNET_HDR` set and `TUNSETVNETHDRSZ` at 12, every frame transferred through the TAP fd is prefixed with a `virtio_net_hdr_v1`:

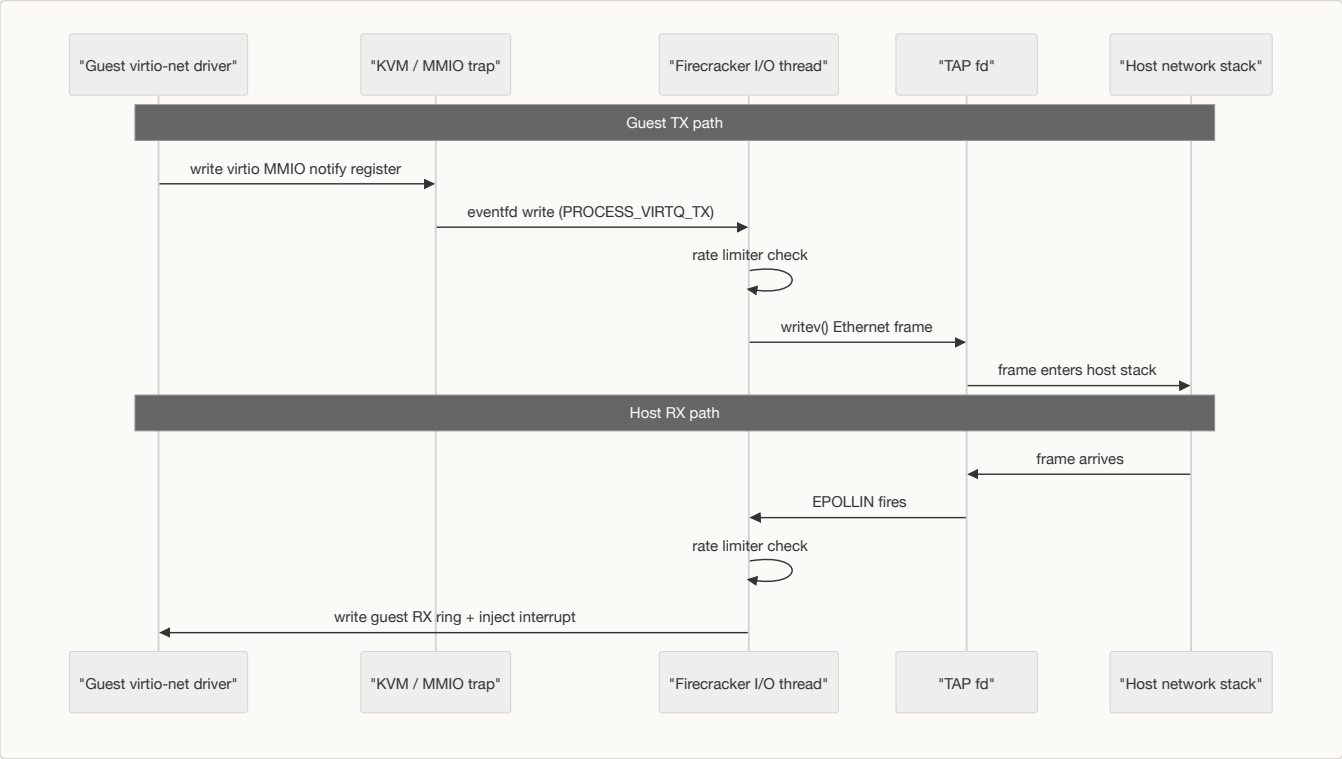
Field	Type	Offset	Meaning
`flags`	u8	0	`VIRTIO_NET_HDR_F_NEEDS_CSUM` = 1, `F_DATA_VALID` = 2
`gso_type`	u8	1	`GSO_NONE` =0, `GSO_TCPV4` =1, `GSO_UDP` =3, `GSO_TCPV6` =4, `GSO_ECN` =0x80
`hdr_len`	u16	2	Transport header length
`gso_size`	u16	4	MSS for segmentation
`csum_start`	u16	6	Byte offset to the checksum field
`csum_offset`	u16	8	Offset to the checksum within the segment
`num_buffers`	u16	10	Number of merged receive buffers

`VIRTIO_NET_F_MRG_RXBUF` (bit 15) is always advertised, which activates the `num_buffers` field and allows a single incoming frame to span multiple guest ring buffers. `VIRTIO_NET_F_MQ` (bit 22) is explicitly not advertised; Firecracker is single-queue only, and `IFF_MULTI_QUEUE` (Linux 3.8+) is defined in the generated bindings but unused.

The packet flows, written out concretely:

Guest TX (guest to host): the guest driver posts a descriptor chain to queue 1 and writes the MMIO notify register; KVM writes the bound eventfd; the epoll loop wakes on `PROCESS_VIRTQ_TX`; the VMM pops the head descriptor, checks the rate limiter (ops bucket first, then bytes bucket), copies scatter-gather data out of guest memory into a TX buffer, and calls `libc::writev()` on the TAP fd, delivering the Ethernet frame to the host kernel's network stack. It then signals the used ring to the guest.

Host to guest RX: `EPOLLIN` fires on the TAP fd; the VMM checks the RX rate limiter (if throttled, it unregisters the TAP fd from epoll until the timer fires); `libc::readv()` fills scatter-gather buffers from the fd; the buffers are written into guest memory; the VMM updates the used ring and injects an interrupt into the guest.



Connecting the TAP to the World

Firecracker supports only the TUN/TAP backend; every networking topology is built outside the VMM. The operator is responsible for creating the TAP device and connecting it to the rest of the host before the microVM boots. Three topologies cover the common cases.

Routed NAT (Point-to-Point)

The minimal and most common setup assigns a `/30` subnet to each TAP device, giving two usable addresses: one on the host-side TAP interface and one configured inside the guest. The host IP-forwards between the TAP and the outbound interface and masquerades the guest traffic. This is the topology Firecracker's `docs/network-setup.md` uses as its starting example, and it scaled to several thousand simultaneous microVMs in the original Firecracker demo.

> ***Root required.*** *The commands below create a TAP device and modify the host routing table. Run them as root or with* `CAP_NET_ADMIN`.

```
# Create the TAP device and assign the host-side address
ip tuntap add tap0 mode tap
ip addr add 172.16.0.1/30 dev tap0
ip link set tap0 up

# Enable IP forwarding
echo 1 > /proc/sys/net/ipv4/ip_forward

# Masquerade guest traffic leaving on eth0 (nftables)
nft add table ip firecracker
nft add chain ip firecracker postrouting \
    '{ type nat hook postrouting priority 100 ; }'
nft add rule ip firecracker postrouting \
    ip saddr 172.16.0.2 oifname eth0 counter masquerade
nft add chain ip firecracker filter \
    '{ type filter hook forward priority 0 ; }'
nft add rule ip firecracker filter \
    iifname tap0 oifname eth0 accept
```

Inside the guest, the equivalent with `iproute2`:

```
ip addr add 172.16.0.2/30 dev eth0
ip link set eth0 up
ip route add default via 172.16.0.1 dev eth0
```

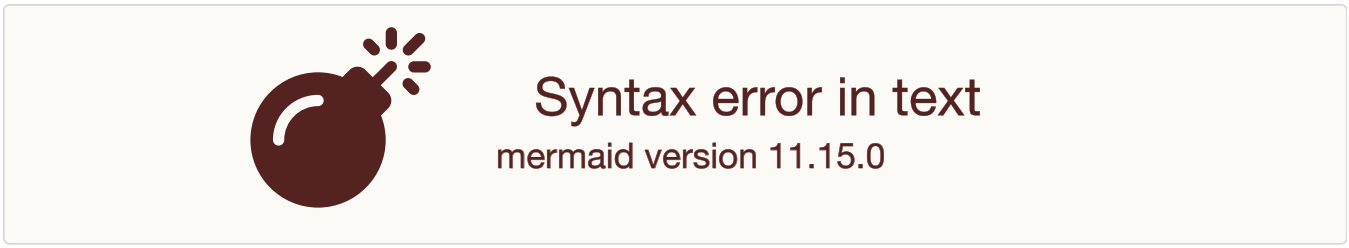
For N microVMs, use sequential `/30` subnets starting at `172.16.0.0/16`. For zero-based ordinal O, the host TAP address is `172.16.[(4*O+1)/256].[(4*O+1)%256]` and the guest address is `172.16.[(4*O+2)/256].[(4*O+2)%256]`. Each TAP stays in its own `/30` broadcast domain, which means no lateral traffic between VMs without routing.

Many operators skip the `iproute2` step entirely and pass the guest IP configuration through the Linux kernel's `ip=` boot parameter, documented in `Documentation/admin-guide/nfs/nfsroot.rst`. The full 10-field positional format:

```
ip=<client-IP>:<server-IP>:<gw-IP>:<netmask>:<hostname>:<device>:<autoconf>:<dns0-IP>:
<dns1-IP>:<ntp0-IP>
```

All fields are optional; trailing colons for omitted fields can be dropped.

For a static Firecracker guest with no server, no hostname, and no autoconf:



The kernel performs the equivalent of `ip addr add`, `ip link set`, and `ip route add default` during boot initialization, before the init process starts. No `iproute2` package and no DHCP client need to be in the rootfs. This is particularly useful with the minimal disk images that Firecracker's fast-boot architecture encourages.

When using Firecracker's official getting-started rootfs, the guest MAC must follow the form `06:00:AC:10:00:02`, where the last four octets encode the guest IPv4 address in hex (`AC:10:00:02` = `172.16.0.2`). This is a convention of that rootfs's init system, not a requirement of Firecracker itself. The API field `guest_mac` is optional; if omitted, the guest kernel generates a random MAC at startup.

Bridge (L2 Multi-VM)

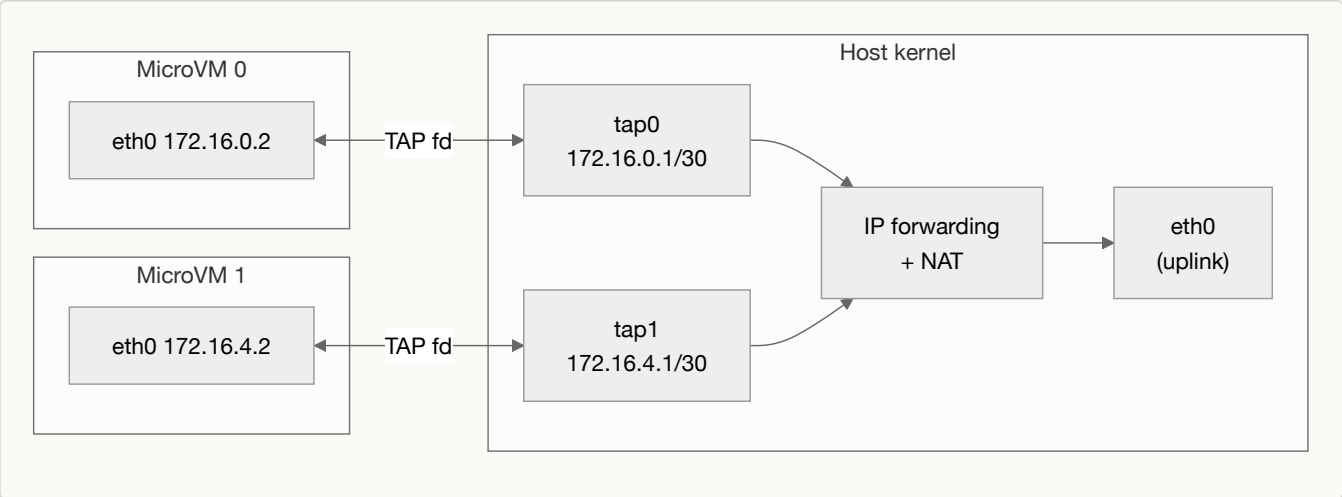
A Linux bridge provides layer-2 connectivity across multiple TAP devices, putting several microVMs on the same broadcast domain:

> ***Root required.*** *The commands below create a bridge and add a TAP to it, modifying the host network topology.*

```
ip link add name br0 type bridge
ip link set dev tap0 master br0
ip link set br0 up
```

With a bridge, the kernel bridge layer handles MAC learning and frame forwarding. Intra-VM communication requires no NAT — two guests on the same bridge communicate directly through the bridge's MAC table. A masquerade rule on the bridge interface still covers external access.

The routed and bridge topologies are the two extremes of isolation. A `/30` point-to-point gives each VM the maximum isolation short of no connectivity at all; a bridge maximizes intra-VM connectivity at the cost of a shared L2 domain. Routed NAT maximizes inter-VM isolation; a bridge gives each VM a shared L2 domain, which production deployments such as Firecracker on AWS Lambda avoid in favor of higher-level service meshes for inter-function traffic.



The Security Boundary

Firecracker performs no network traffic filtering. Its `docs/design.md` states plainly: "all outbound network traffic data is copied by the Firecracker I/O thread from the emulated network interface to the backing host TAP device." Filtering is the host operator's responsibility. In practice that means nftables or iptables rules on the host, applied to the TAP interface, before traffic reaches the bridge or the routing table.

Rate Limiting at the VMM Edge

The problem rate limiting solves for microVMs is the same one it solves for containers: a single noisy tenant can starve every other workload on the same host.

Token Bucket in VMM Userspace

Firecracker rate limiting executes entirely in VMM userspace. There is no kernel qdisc, no netfilter rule, and no traffic control class. The I/O thread applies token-bucket checks synchronously, in the same epoll handler that moves data between the virtqueue ring buffers and the TAP fd. A frame that fails the bucket check never reaches the TAP fd — the kernel never sees it.

Each network interface in the Firecracker API accepts two optional `RateLimiter` objects, one for RX and one for TX. Each `RateLimiter` holds two independent `TokenBucket` configurations:

- **bandwidth**: the unit is bytes; limits throughput.
- **ops**: the unit is packets; limits packet rate independently of size.

Both run simultaneously. The ops bucket is checked first (`consume(1, TokenType::Ops)`); if it passes, the bytes bucket is checked (`consume(frame_size, TokenType::Bytes)`). If the bytes check fails after the ops check succeeded, the ops token is manually replenished (`manual_replenish(1, Ops)`) to keep the two buckets consistent. Both buckets must pass for a frame to proceed.

`TokenBucket` has three fields:

Field	Type	Notes
`size`	int64	Bucket capacity and initial budget
`refill_time`	int64 (ms)	Time to refill one full bucket
`one_time_burst`	int64	Non-replenishing burst, consumed before the main budget

The bucket starts full (`budget = size`). The `one_time_burst` is consumed first and does not replenish after draining — it is a startup grace period, not a sustained burst allowance. Setting `size = 0` or `refill_time = 0` disables that limiter.

The refill formula, from `src/vmm/src/rate_limiter/mod.rs`:

```
refill_tokens = (time_delta_ns * size) / (refill_time_ms * 1_000_000)
```

To avoid integer overflow for large `size` values, the implementation pre-divides `size` and `complete_refill_time_ns` by their GCD (Euclidean algorithm) and stores the reduced pair as `processed_capacity` / `processed_refill_time`. The refill polling interval is `REFILL_TIMER_DURATION = 100 ms`, driven by a `TimerFd` per `RateLimiter` that is armed when the bucket hits empty and cleared by `event_handler()` when the timer fires.

The `reduce()` method returns one of three variants. `BucketReduction::Success` means enough tokens are available and the packet proceeds. `BucketReduction::Failure` means the bucket is dry; the timer is armed, draining halts, and — for RX — the TAP fd is unregistered from epoll until the refill fires. `BucketReduction::OverConsumption(f64)` is the interesting edge case: the frame is larger than the full bucket capacity. The VMM lets it through (dropping an oversized frame would be worse) but arms the timer for `ratio * refill_time` ms to compensate. The `rx_rate_limiter_throttled` metric is incremented on throttle.

Rate limiter configuration is pre-boot via `PUT /network-interfaces/{iface_id}`. Live reconfiguration on a running microVM is available through `PATCH /network-interfaces/{iface_id}`, introduced in Firecracker v0.15.0.

Comparison With CNI Bandwidth Shaping

Container networking shapes traffic through the kernel qdisc scheduler. The CNI bandwidth meta-plugin adds a `tbf` (token bucket filter) qdisc to the host-side veth using `RTM_NEWQDISC` netlink calls. For egress shaping (pod-to-host direction), it attaches an `ingress` qdisc (`handle ffff:`) to the host-side veth, installs a U32 filter with a `MirredAction` (`TCA_EGRESS_REDIR`) that redirects matching traffic to an IFB device named `bwp<hash>` (max 15 chars), and adds a `tbf` root qdisc on the IFB. For ingress shaping (host-to-pod), a `tbf` root qdisc goes directly on the host-side veth. The CNI binary exits after setup; enforcement is entirely in the kernel packet scheduler.

The `tbf` qdisc parameters: `Rate` (bytes/s, computed as `ingressRate / 8` since the CNI config specifies bits), `Limit` (derived from rate, burst, and a hardcoded `latencyInMillis = 25`), and `Buffer` (burst in kernel tick units). Burst values in the CNI config are in bits; the burst/8 value must be less than 2^32 bytes.

The architectural split:

Property	Firecracker rate limiter	CNI bandwidth plugin
Enforcement point	VMM userspace I/O thread (epoll)	Linux kernel qdisc on host-side veth
Algorithm	Custom Rust token bucket	Kernel `tbf` qdisc (`net/sched/sch_tbf.c`)
Granularity	Per-NIC, per-direction, dual bucket (bytes + ops)	Per-NIC, per-direction, bytes only
Burst mechanism	`one_time_burst` (non-replenishing)	`burst` parameter (bits, converted to buffer ticks)
Refill timer	100 ms `TimerFd` in VMM userspace	Kernel packet scheduler tick
Live reconfiguration	Yes, PATCH since v0.15.0	No — requires deleting and recreating the pod
Isolation boundary	Before packet reaches TAP fd or kernel	After packet exits the container namespace

The `tbf` qdisc only sheds bytes. Firecracker's dual-bucket design lets operators cap packet rate (ops/s) independently from byte throughput, which is useful for controlling CPU cost from small-packet floods — a stream of 64-byte UDP packets consumes almost no bandwidth but can saturate the VMM's I/O thread with interrupt processing. Setting an ops limit sheds packets before the VMM copies them, directly capping interrupt load.

The enforcement point is the other decisive difference. The Firecracker rate limiter intercepts traffic before the kernel ever sees the frame. The CNI/qdisc path enforces limits in the kernel scheduler after the packet has already crossed the veth pair into the host network stack. For a microVM, there is no veth pair and no shared namespace to cross — the TAP fd is the only crossing point, and the rate limiter sits between the virtqueue and the fd.

MMDS: In-Process Metadata Service

A microVM that boots from a minimal rootfs with no DHCP client needs another way to receive its per-instance configuration: its IP address, its TLS certificate, its role credentials, the bootstrap data that tells its init process what to run. The EC2 Instance Metadata Service (IMDS) solved this problem when EC2

launched by placing a magic link-local address (`169.254.169.254`) on every instance that routes to the hypervisor's management plane. Firecracker embeds the equivalent: the **MicroVM Metadata Service** (MMDS).

Architecture: Dumbo Inside the Data Path

MMDS is not a sidecar process and not a separate network path. It is three components embedded directly in the Firecracker VMM:

1. A **host-side HTTP API handler** that lets the operator populate a per-VM JSON data store before or after boot.
2. A **global JSON data store** (`serde_json::Value` , default size limit 51,200 bytes, configurable with `--mmds-size-limit`).
3. **Dumbo**, a minimalist TCP/IPv4/ARP network stack implemented in Rust, embedded in the virtio-net data path.

Dumbo intercepts frames between the guest's virtio ring buffers and the TAP fd. For each guest-to-host frame on an MMDS-enabled interface, the VMM applies a heuristic: if the frame could be an ARP request for the MMDS IP or an IPv4 packet destined to the MMDS IP, Dumbo handles it and the frame never reaches the TAP fd. Otherwise the frame is forwarded normally. The heuristic has no false negatives — a frame that might be for MMDS is never incorrectly forwarded to the host.

MMDS is disabled by default. Enabling it requires associating it with one or more network interfaces via `PUT /mmds/config` . The `network_interfaces` field (array of interface ID strings) controls which TAP devices Dumbo monitors; frames arriving on non-associated interfaces pass through unmodified.

MMDS answers at `169.254.169.254` by default, overridable with any valid link-local IPv4 via `ipv4_address` in `MmdsConfig` . The hardcoded Dumbo source MAC is `06:01:23:45:67:01` , used in ARP replies and as the source MAC on all outgoing TCP segments. TTL on all MMDS outgoing packets is 1. The guest must add a host route for the MMDS IP:

```
ip route add 169.254.169.254 dev eth0
```

Without a DHCP client this route must be injected by the init system, an in-guest script, or through the kernel `ip=` parameter using a supplementary static route mechanism.

```
flowchart LR
    subgraph guest["Guest kernel"]
        drv["virtio-net driver"]
    end

    subgraph vmm["Firecracker VMM process"]
        vq["Virtio ring buffers"]
        dumbo["Dumbo\n(ARP + TCP/IP stack)"]
        store["JSON data store"]
        tap["TAP fd"]
    end

    hostapi["Host API\nPUT /mmds"]

    drv <-->|"MMIO + DMA"| vq
    vq --> dumbo
    dumbo -->|"frame for 169.254.169.254"| store
    dumbo -->|"all other frames"| tap
    hostapi --> store
```

Dumbo's Constraints

Dumbo is a deliberately narrow implementation. The design document enumerates its limitations explicitly:

- No 802.1Q VLAN tag support; tagged frames pass through to the TAP fd unexamined.
- No IP fragmentation reassembly; fragmented packets are treated as independent datagrams.
- Only EtherType 0x0806 (ARP) and 0x0800 (IPv4) are processed; IPv6 is dropped.
- Minimal TCP: flow control only, no congestion control, no support for most TCP options.
- At most one pending HTTP response per TCP connection.
- If a guest request exceeds the fixed receive buffer, the connection is reset.

When an ARP request targeting the MMDS address arrives, Dumbo records it (retaining only the most recent). On the next available slot in the guest receive ring, it sends an ARP reply with source MAC `06:01:23:45:67:01` before serving any TCP segments. This ordering guarantees the guest's ARP table has the MMDS MAC before any HTTP connection attempt.

Host API and Data Store

Four endpoints manage MMDS:

Endpoint	Method	Purpose
`PUT /mmds/config`	PUT	Set MMDS version, IPv4 address, and allowed interface IDs (pre-boot)
`PUT /mmds`	PUT	Replace entire data store with any valid JSON
`PATCH /mmds`	PATCH	Partial update via JSON Merge Patch (RFC 7396)
`GET /mmds`	GET	Retrieve full data store from host side

The data store accepts any valid JSON. The MMDS version, network configuration, and IPv4 address are preserved across snapshot and restore. The data store itself is not persisted across snapshots — a snapshot of a running VM clears it — to avoid leaking per-VM secrets to clones derived from the same snapshot.

V1 and V2

MMDS has two protocol versions. V1 is stateless and deprecated; it is scheduled for removal in the next major Firecracker release. V2 is modeled after AWS IMDSv2 and is the version new deployments should use.

V2 session flow: the guest first obtains a session token:

```
PUT http://169.254.169.254/latest/api/token
X-metadata-token-ttl-seconds: 21600
```

Dumbo responds with a token string. TTL is between 1 and 21600 seconds (6 hours). The `X-Forwarded-For` header must not be present in the token request; its presence causes a 400 to be returned, which closes the same SSRF attack surface that IMDSv2 targets. Subsequent data requests supply the token:

```
GET http://169.254.169.254/latest/meta-data/ami-id
X-metadata-token: <token>
```

An invalid or expired token returns 401 Unauthorized. V2 also accepts the AWS header names (`X-aws-ec2-metadata-token-ttl-seconds` , `X-aws-ec2-metadata-token`) to allow unmodified EC2 IMDS clients to run against a Firecracker MMDS without modification.

V1 skips the token step entirely. Requests arrive without authentication and Dumbo responds. Metrics `mmds.rx_invalid_token` and `mmds.rx_no_token` are incremented on V1 requests so operators can track migration progress, but no enforcement occurs.

Addressing the Data Store

Resources are addressed by JSON Pointer (RFC 6901) as the URI path. For a data store containing:

```
{
  "latest": {
    "meta-data": {
      "ami-id": "ami-12345678",
      "local-ipv4": "172.16.0.2"
    }
  }
}
```

the guest fetches `/latest/meta-data/ami-id` and receives `ami-12345678`. Two response formats are available: `Accept: application/json` returns JSON; `Accept: plain/text` (or no header) returns IMDS text format, where object keys are separated by newlines and objects are represented with a trailing `/`, matching AWS EC2 IMDS behavior. Note that `plain/text` is a Firecracker-specific token, not the RFC 2045-compliant `text/plain`; this is an intentional Firecracker quirk, not a typo. Setting `imds_compat: true` in `MmdsConfig` forces IMDS format regardless of the `Accept` header, enabling unmodified EC2 IMDS clients. JSON types that have no IMDS text representation (numbers, arrays, booleans) return 501 when IMDS format is requested.

Why Not a veth?

The last question this chapter should answer is why Firecracker does not use the same veth-plus-CNI architecture that container runtimes use. The answer is the second kernel.

A container is a process in a separate Linux network namespace. Its veth pair is a kernel object that tunnels between two namespaces within the same kernel. The kernel bridge or route table handles switching between them. No userspace process mediates every packet; the kernel does it directly.

A microVM runs a separate kernel. There is no shared kernel object that spans both kernels — any communication must cross from the guest kernel into the VMM process and then back into the host kernel. The TAP fd is the crossing point. Firecracker reads frames from the guest virtqueue (shared memory that KVM maps into both address spaces), converts them into `writev()` calls on the TAP fd, and lets the host kernel handle them from there. The veth pair is a kernel-to-kernel shortcut that simply does not exist across a VM boundary.

The CNI model could be grafted on top of the TAP device — some orchestrators do exactly that, running CNI plugins that configure the host-side TAP rather than a veth. But the virtio-net device, the epoll event loop, and the rate limiter always remain between the guest and whatever the host side is.

Sources And Further Reading

- Linux TUN/TAP documentation: https://docs.kernel.org/networking/tuntap.html
- Linux `if_tun.h` kernel uapi: https://github.com/torvalds/linux/blob/master/include/uapi/linux/if_tun.h

- Linux `virtio_net.h` kernel uapi: https://github.com/torvalds/linux/blob/master/include/uapi/linux/virtio_net.h
- OASIS virtio 1.2 CS01: https://docs.oasis-open.org/virtio/virtio/v1.2/cs01/virtio-v1.2-cs01.html
- Firecracker `tap.rs`: https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/devices/virtio/net/tap.rs
- Firecracker `device.rs` (net): https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/devices/virtio/net/device.rs
- Firecracker `event_handler.rs` (net): https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/devices/virtio/net/event_handler.rs
- Firecracker `mod.rs` (net): https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/devices/virtio/net/mod.rs
- Firecracker `rate_limiter/mod.rs`: https://github.com/firecracker-microvm/firecracker/blob/main/src/vmm/src/rate_limiter/mod.rs
- Firecracker OpenAPI spec: https://raw.githubusercontent.com/firecracker-microvm/firecracker/main/src/firecracker/swagger/firecracker.yaml
- Firecracker `network-setup.md`: https://github.com/firecracker-microvm/firecracker/blob/main/docs/network-setup.md
- Firecracker `design.md`: https://github.com/firecracker-microvm/firecracker/blob/main/docs/design.md
- Firecracker `mmds-design.md`: https://github.com/firecracker-microvm/firecracker/blob/main/docs/mmds/mmds-design.md
- Firecracker `mmds-user-guide.md`: https://github.com/firecracker-microvm/firecracker/blob/main/docs/mmds/mmds-user-guide.md
- CNI bandwidth plugin `main.go`: https://github.com/containernetworking/plugins/blob/main/plugins/meta/bandwidth/main.go
- CNI bandwidth plugin `ifb_creator.go`: https://github.com/containernetworking/plugins/blob/main/plugins/meta/bandwidth/ifb_creator.go
- CNI bandwidth plugin docs: https://www.cni.dev/plugins/current/meta/bandwidth/
- Linux kernel NFS root docs (`ip=` parameter): https://docs.kernel.org/admin-guide/nfs/nfsroot.html

Chapter 22: MicroVMs As Containers

Every runtime that runs a container on Linux eventually calls `execve`. The process on the other end of that call is still subject to the same host kernel, the same syscall table, and the same kernel bugs as everything else on the node. The namespace and cgroup boundaries are real, but they are software: a kernel vulnerability can walk past them. The microVM exists precisely to put a hardware boundary between the workload and the host — a second kernel, a separate address space enforced by EPT rather than by Linux's own memory management. The question this chapter answers is what happens when you want *both*: the operational interface of a container — the OCI image, the CRI API, the Kubernetes pod spec — wrapped around the isolation of a virtual machine.

Three projects answer that question in different ways. **firecracker-containerd** replaces the runC shim with a Firecracker VMM while keeping the containerd daemon and its entire API surface intact. **Kata Containers** does the same at the CRI level, routing every Kubernetes pod into a dedicated VM, transparent to kubelet. **flintlock** takes the idea one level up: instead of running a container inside a microVM, it provisions a microVM that *is* a Kubernetes node, managed through an OCI-friendly API and controlled by a Cluster API provider. The first two care about what runs inside the VM; the third cares about the VM as the unit of infrastructure.

The place where all three projects touch the containerd book is the containerd shim v2 protocol. Understanding that handoff is the key to understanding everything else.

The Shim v2 Protocol As The Pivot Point

Three radically different systems — runC containers, Firecracker microVMs, and QEMU-backed Kata pods — appear identical to containerd above a single protocol boundary. That boundary is the shim v2 protocol, and understanding it is what makes the rest of this chapter legible.

When containerd starts a task, it does not call runC directly. It finds a shim binary, forks it, and from that point forward speaks the `containerd.task.v2.Task` ttrpc service defined in `api/runtime/task/v2/shim.proto`. The shim is responsible for everything below that service boundary: creating the container, managing its lifecycle, mounting its rootfs, and forwarding stdio. Containerd does not care what the shim actually does. It calls `Create`, `Start`, `Kill`, and `Delete` over ttrpc and trusts the shim to make it happen.

The runtime name in the container spec determines which shim binary runs. Containerd takes the last two dot-separated components of the runtime name, replaces every `.` with `-` within those components, and prepends `containerd-shim-`. The runtime name `io.containerd.runc.v2` yields `containerd-shim-runc-v2`; `aws.firecracker` yields `containerd-shim-aws-firecracker`; `io.containerd.kata.v2` yields `containerd-shim-kata-v2`. Starting with containerd 1.6.0, the binary path can be given directly instead of relying on the derivation.

The shim service has seventeen RPC methods: `State`, `Create`, `Start`, `Delete`, `Pids`, `Pause`, `Resume`, `Checkpoint`, `Kill`, `Exec`, `ResizePty`, `CloseIO`, `Update`, `Wait`, `Stats`, `Connect`, and `Shutdown`. The shim must publish four events in strict order — `TaskCreateEventTopic`, `TaskStartEventTopic`, `TaskExitEventTopic`, `TaskDeleteEventTopic` — and is responsible for mounting the container rootfs into the `rootfs/` subdirectory of the OCI bundle path it receives in `CreateTaskRequest`. In containerd 2.3 and later, the shim reads a protobuf `BootstrapParams` message from stdin and writes a `BootstrapResult` to stdout. Legacy shims write a JSON object like `{"version": 2, "address": "/path", "protocol": "grpc"}` to stdout.

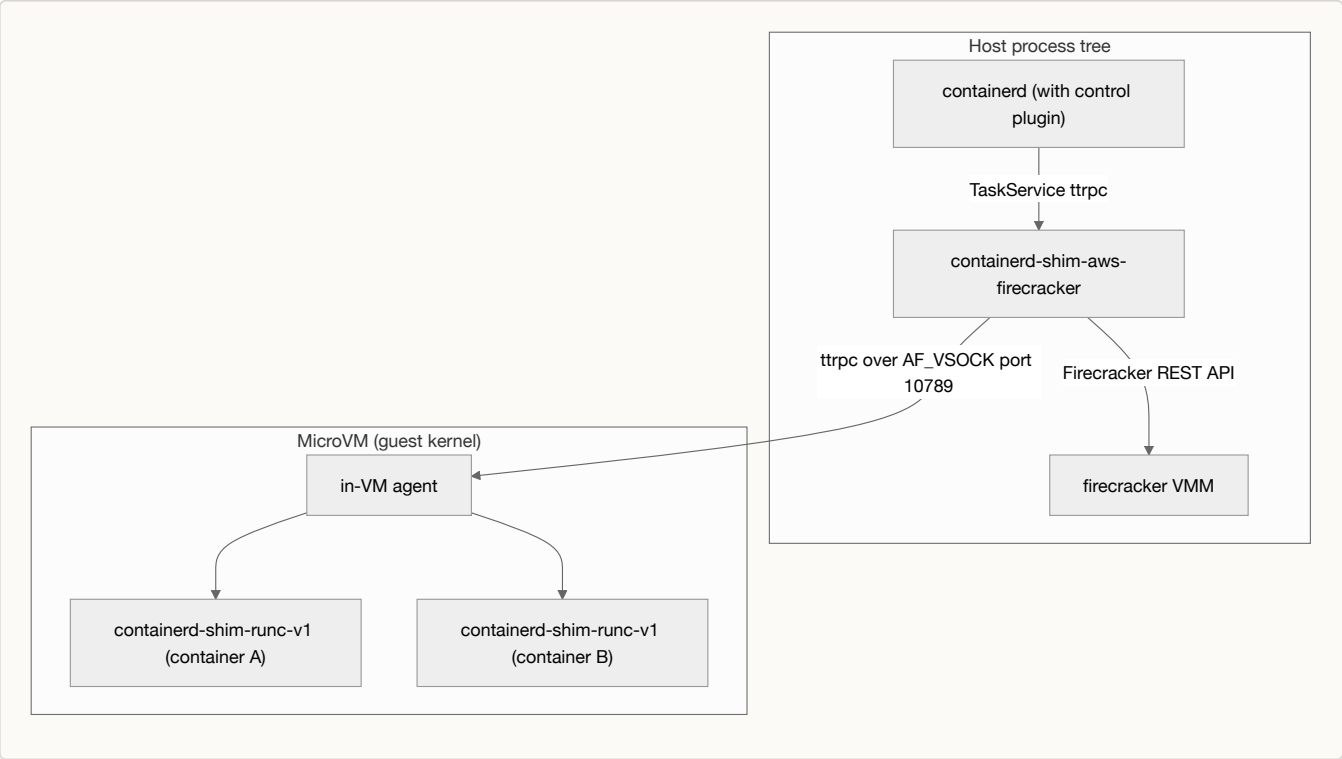
Everything above the shim — container creation, image pulling, snapshot management, event routing, the CRI API — is identical whether the shim is `containerd-shim-runc-v2` or `containerd-shim-aws-firecracker`. The hardware boundary lives entirely below the seventeen ttrpc methods.

firecracker-containerd

One Shim, One VM, Many Containers

The naive mapping from containerd's task model to Firecracker would fork one VMM per container. That is the wrong answer: each Firecracker process carries its own memory overhead, its own guest kernel boot, and its own vCPU threads. firecracker-containerd instead multiplexes: one shim process manages all containers within a single microVM. The first `Create` call for a given VM ID starts the shim and boots the VMM; subsequent tasks in the same VM reuse the running instance. Containerd locates an existing shim for a VM via pid and socket files at `/var/lib/firecracker-containerd/shim-base/<vm_id>/`. If the client supplies no VM ID, the shim generates a UUID v4, giving the default topology of one container per VM.

The project ships four components: the **control plugin** (compiled directly into a specialized `containerd` binary — not a standalone process or a socket-based plugin), the **host-side runtime shim** (`containerd-shim-aws-firecracker`), the **in-VM agent**, and a root filesystem image builder. The control plugin manages microVM lifecycle and exposes an API modeled on the Firecracker lifecycle, defined in `proto/firecracker.proto`. Its configuration types — `FirecrackerMachineConfiguration` (fields: `CPUTemplate`, `HtEnabled`, `MemSizeMib`, `VcpuCount`), `CNIConfiguration` (`NetworkName`, `InterfaceName`, `BinPath`, `ConfDir`, `CacheDir`, `Args`), and `FirecrackerDriveMount` (`HostPath`, `VMPath`, `FilesystemType`, `Options`, `RateLimiter`, `IsWritable`, `CacheType`) — let the caller declare the VM's hardware before any container is created.



VSOCK: The Control Plane Across The Hardware Boundary

The host shim connects to the in-VM agent over AF_VSOCK using the ttrpc protocol. Two port constants govern this, both confirmed in `runtime/service.go`: `defaultVsockPort` is `10789`, used for the ttrpc control channel, and `minVsockIOPort` is `11000` (a `uint32`), used as the base port for stdio multiplexing. Stdio for each container in the VM occupies three consecutive ports starting at `minVsockIOPort`: stdin on port 11000, stdout on 11001, stderr on 11002; a second container gets 11003–11005, and so on.

The VSOCK device in the Firecracker Go SDK is configured with `ID: "agent_api"`, a `GuestCid` cast from `uint32` to `int64`, and a `UdsPath` pointing to a host-side Unix domain socket. Firecracker's VSOCK multiplexing protocol adds one indirection: to reach a given port in the guest, the host connects to the UDS path and sends the string `"CONNECT <port_num>\n"`; Firecracker responds with `"OK <assigned_hostside_port>\n"`. Guest-to-host connections work in the opposite direction: the guest targets CID 2 (the host CID), and Firecracker forwards the connection to `<uds_path>_<port_number>` on the host.

Block Devices, Not Filesystems

The snapshotter is an out-of-process gRPC proxy plugin that implements containerd's snapshotter API. firecracker-containerd ships two implementations. The `naive` snapshotter does a full file copy per snapshot — useful as a proof of concept but produces no deduplication. The `devmapper` snapshotter uses device-mapper thin provisioning for copy-on-write layering, matching what production systems need. The

thin pool is named `fc-dev-thinpool` with `base_image_size = "10GB"` and metadata rooted at `/var/lib/firecracker-containerd/snapshotter/devmapper`. Containerd identifies it as `io.containerd.snapshotter.v1.devmapper` in `config.toml`.

The important design choice is that snapshots are exposed to Firecracker as **block devices**. There is no virtiofs, no 9p, no filesystem-level sharing across the hardware boundary. The host exports a device-mapper thin volume; the guest kernel mounts it directly. This is a deliberate constraint: virtiofs requires a shared memory region and a dedicated `virtiofsd` process, introducing host-kernel surface area that firecracker-containerd's threat model rules out.

Firecracker has no hot-plug for block devices. Every drive must be declared before the VM boots. The runtime solves this with **drive stub pre-allocation**: before boot it attaches sparse stub files (minimum 128 bytes) or `/dev/null` aliases to reserve drive IDs. When a container is created and its snapshot is ready, the runtime calls `PATCH /drives/{drive_id}` — the Firecracker REST endpoint `PatchGuestDriveByID` — to swap the stub for the real block device. The `mountDrives` function at around line 821 of `runtime/service.go` iterates `s.driveMountStubs` and calls `PatchAndMount` on each before the in-VM agent becomes available. Inside the VM, the agent correlates drives to containers either by position (`lsblk` sorted by `MAJOR:MINOR`) or by content (a drive ID written into the stub file that persists on the real device).

The Network: tc-redirect-tap

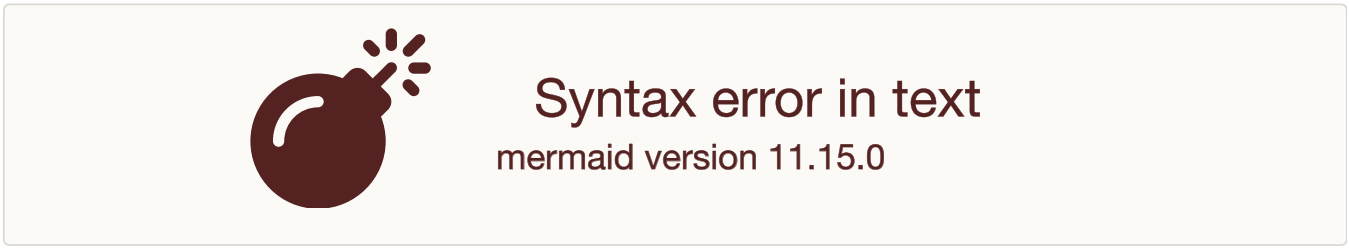
Network configuration in firecracker-containerd happens at VM-creation time, not per-container. The caller passes a `CNIConfiguration` to `CreateVM` ; the runtime runs a CNI chain of three plugins. First, `ptp` creates a veth pair inside a dedicated network namespace. Second, `host-local` handles IPAM. Third, `tc-redirect-tap` — a custom Firecracker plugin — creates a TAP device inside that same network namespace and installs Linux TC (Traffic Control) U32 filters to redirect traffic bidirectionally between the TAP and the veth at the ingress/egress filter level.

This is the same TAP-plus-TC pattern Chapter 21 describes for raw Firecracker networking, but the CNI chain wraps it so the IP address and network plumbing come from the standard CNI IPAM path rather than being hardcoded. The Firecracker Go SDK creates and manages the network namespace, invokes CNI within it, starts the VMM inside that namespace, and handles CNI teardown on VM termination. Inside the guest, the IP and DNS arrive via Linux kernel boot parameters; the guest `/etc/resolv.conf` is symlinked to `/proc/net/pnp` . The sample network is named `"fcnet"` with interface name `"veth0"` ; CNI config lives under `/etc/cni/conf.d` and plugin binaries under `/opt/cni/bin` .

The In-VM Agent And Boot Args

The in-VM agent must be embedded in the root filesystem image and configured to start on boot. Once running, it accepts control instructions from the host shim over the VSOCK ttrpc channel, invokes standard Linux containers via `containerd-shim-runc-v1` (runC v1, not v2) inside the VM, emits events

and metrics back to the shim, and proxies stdio over the per-container VSOCK port triples. The production kernel command line used with firecracker-containerd is:



The `init=/sbin/overlay-init` argument indicates a custom init binary that handles the overlay filesystem setup before handing control to the agent — the agent may be `overlay-init` itself or a child it launches. The `pci=off nomodules` arguments reflect Firecracker's stripped device model: without PCI and with a pre-configured driver set baked into the kernel, module loading is neither needed nor possible.

> *The containerd socket for firecracker-containerd lives at `/run/firecracker-containerd/containerd.sock`, with state at `/run/firecracker-containerd` and content root at `/var/lib/firecracker-containerd/containerd`. The CRI plugin is explicitly disabled in `config.toml`: `disabled_plugins = ["io.containerd.grpc.v1.cri"]`. This is not the containerd instance kubelet talks to; it is a dedicated daemon for the Firecracker integration.*

Kata Containers

The Same Shim Protocol, A Different VM Strategy

Kata Containers ships as a single binary, `containerd-shim-kata-v2`, derived from the runtime handler string `io.containerd.kata.v2` by the same naming convention. Like firecracker-containerd, one shim instance manages all containers in a single pod's VM — there is no per-container shim fork. Unlike firecracker-containerd, Kata targets the Kubernetes CRI path directly. Kubernetes RuntimeClass (stable since Kubernetes 1.12) routes pods to `containerd-shim-kata-v2` via `runtimeClassName: kata` in the pod spec. The CRI runtime signals sandbox versus container membership to the shim via an OCI annotation: `io.kubernetes.cri.container-type` when containerd is the CRI runtime (the primary path in this book), or `io.kubernetes.cri-o.ContainerType` when CRI-O is. Both carry values `sandbox` or `container`.

The mapping is clean: each Kubernetes pod becomes one Kata VM, and each container in that pod becomes one process inside that VM. The API boundary above the shim — the CRI `RunPodSandbox` and `CreateContainer` calls — remains identical to what kubelet sends for a runC pod. The VM appears nowhere in the Kubernetes API.

kata-agent: Rust, PID 1, 43 Methods

The in-VM agent for Kata, `kata-agent`, has been written in Rust since Kata 2.0. It runs as a long-running process inside the VM and is responsible for the full container lifecycle within that VM. When the guest image is an initrd, `kata-agent` runs as PID 1 at `/sbin/init`, built with `AGENT_INIT=yes`. When the guest image is a rootfs, systemd starts as PID 1 and launches `kata-agent` as a systemd service. The agent listens on vsock address `vsock://-1:1024` (`VMADDR_CID_ANY` on port 1024), accepting connections regardless of the guest CID the hypervisor assigns. A separate configurable vsock port (the `agent.log_vport` option) is reserved for log forwarding; it defaults to 0 and must be set explicitly.

The agent exposes the `AgentService` ttrpc service defined in `src/libs/protocols/protos/agent.proto` (package `grpc`). In the current main branch it has 43 RPC methods. The categories span sandbox lifecycle (`CreateSandbox`, `DestroySandbox`), container lifecycle (`CreateContainer`, `StartContainer`, `StatsContainer`, and four more), process operations (`ExecProcess`, `SignalProcess`, `WaitProcess`), I/O, networking (`UpdateInterface`, `UpdateRoutes`, `GetIPTables`, `SetIPTables`, and three more), storage (`GetVolumeStats`, `ResizeVolume`, `AddSwap`), guest introspection (`GetGuestDetails`, `GetMetrics`, `GetOOMEvent`), hotplug (`OnlineCPUMem`, `MemHotplugByProbe`), and policy.

The breadth reflects Kata's broader scope relative to firecracker-containerd: interface updates, routing changes, iptables management, and memory hotplug are all in-VM operations the shim must be able to drive without touching the host kernel.

Before vsock, Kata used a virtio serial port for agent communication and a separate `kata-proxy` process on the host for mux/demux. Vsock eliminated the proxy. The `vhost_vsock` kernel module must be loaded on the host (`CONFIG_VHOST_VSOCK=y`; `sudo modprobe -i vhost_vsock`); the kernel requirement is Linux 4.8 or later.

> ***Before running the above command:*** *loading `vhost_vsock` is a host-kernel operation requiring root. Verify the module is available with `modinfo vhost_vsock` before attempting to load it. On shared hosts or cloud VMs without nested virtualization, the module may not be present.*

OCI Bundle Delivery Over the Wire

With runC, delivering a container's rootfs to the container runtime is a local filesystem operation: the shim mounts the snapshotter's device into the bundle's `rootfs/` directory before calling `runc create`. With Kata, there is a hardware boundary between the shim and the container, so the bundle cannot be mounted locally and then used. Kata handles this in two steps.

The OCI `config.json` is transmitted to `kata-agent` via ttrpc — specifically a `CreateContainerRequest` over vsock — rather than via a filesystem path inside the VM. The container rootfs travels by one of two mechanisms depending on configuration. The default is **virtiofs**: the shim starts a `virtiofsd` daemon on the host (one per VM), exports the snapshotter directory, and the agent mounts the exported directory as the container rootfs inside the VM. The alternative is **virtio-scsi**: when

block-based graph drivers are configured, the block device is hot-plugged into the VM as a virtio-scsi device, appearing as `/dev/sda` or a similar SCSI path. A distinct case applies when using the devmapper snapshotter with Kata-on-Firecracker: there the block device is passed as a virtio-blk drive, visible inside the VM as `/dev/vda`.

The guest OS mini-image — the kernel and minimal userspace that runs inside each Kata VM — is mounted via DAX to avoid double-caching its pages in both the host and guest page caches. QEMU uses an NVDIMM device backed by a file (the guest sees it as `/dev/pmem*`); Cloud Hypervisor uses an emulated Persistent Memory (PMEM) device; both mount as ext4 inside the VM. virtio-9p is a supported but non-default alternative to virtiofs for rootfs sharing.

```
sequenceDiagram
    participant K as kubelet
    participant C as "containerd CRI"
    participant S as "containerd-shim-kata-v2"
    participant V as virtiofsd
    participant A as kata-agent
    participant R as "runc (inside VM)"

    K->>C: RunPodSandbox
    C->>S: Create (TaskService ttrpc)
    S->>S: boot hypervisor VM
    S->>V: start virtiofsd (one per VM)
    S->>A: CreateSandbox (AgentService vsock:1024)
    K->>C: CreateContainer
    C->>S: Create (container task)
    S->>A: CreateContainer (config.json over vsock)
    A->>V: mount virtiofs export as rootfs
    A->>R: exec container process
    R-->>A: process running
    A-->>S: container started
    S-->>C: task running
    C-->>K: container ready
```

Networking: TC Redirect Without MACVTAP

Hypervisors cannot use a veth interface directly: veth works at the network namespace boundary in the kernel, but the VM's network interface must attach at the TAP layer so the VMM can forward raw Ethernet frames into the guest. Kata's shim creates a TAP interface in the host network namespace and installs TC redirect rules to bridge traffic bidirectionally between the pod's veth interface (which CNI already populated with an IP and routes) and the VM's TAP interface, at the ingress and egress filter level. This replaced an earlier approach that used MACVTAP; the TC redirect approach is more composable with CNI chains and avoids the MAC address management burden MACVTAP introduces.

The flow across a node where kubelet has assigned the pod network via CNI is: CNI populates the veth in the pod's network namespace; the shim bridges traffic from that veth to the TAP; Firecracker or QEMU connects the TAP to a virtio-net device; the guest kernel sees a `eth0` with the CNI-assigned IP. From the pod's perspective — and from kubelet's — the network is indistinguishable from a plain container's. From the kernel's perspective, every packet crosses the hardware boundary of the VM twice.

Four Hypervisors, One Shim

All four VMM backends are invoked through `containerd-shim-kata-v2`; the selection is made in `configuration.toml`. The current releases in the main branch are QEMU v10.2.1 (supporting x86_64, aarch64, ppc64le, s390x, and riscv, with the largest device footprint and broadest architecture coverage), Cloud Hypervisor v51.1 (x86_64 and aarch64, with fine-grained per-thread seccomp and an HTTP OpenAPI management interface), Firecracker v1.12.1 (x86_64 and aarch64, with the most restrictions: no virtiofs, no device hotplug, no VFIO, no CPU or memory hotplug, devmapper snapshotter required), and Dragonball (no separate released version; runs in-process with the `runtime-rs` shim at zero IPC overhead).

Firecracker as a Kata backend is configured via `/etc/kata-containers/configuration-fc.toml`. Because Firecracker does not support virtiofs, the devmapper snapshotter is mandatory and virtio-block is the only rootfs transport. This makes the Kata-on-Firecracker path more constrained than the Kata-on-QEMU path but also closer to what firecracker-containerd does directly.

Kata 3.0 introduced `runtime-rs`, a Rust-based shim with Dragonball as an embedded VMM running in the same process as the shim. Since the VMM and the shim share an address space, the REST API round-trip between them disappears entirely. Kata 4.0 — planned for July 2026 — makes `runtime-rs` the default; the Go runtime enters deprecation at that point, receiving only bug and security fixes with removal no earlier than version 5.0.

Flintlock: The VM As A Kubernetes Node

A Different Level Of Abstraction

firecracker-containerd and Kata Containers both put the microVM inside the container model: the container is the unit of work, and the VM is the isolation mechanism. Flintlock inverts the relationship. The microVM is the unit of infrastructure. Its primary use case is not running application containers but provisioning the nodes those containers will run on — Kubernetes worker and control-plane nodes, managed through an API that speaks OCI image references for kernel and disk images.

Flintlock (`liquidmetal-dev/flintlock`, formerly `weaveworks/flintlock`, MPL-2.0) is a host-level daemon written in Go. It exposes a gRPC service named `MicroVM` in package `microvm.services.api.v1alpha1` with exactly five RPC methods: `CreateMicroVM`, `DeleteMicroVM`, `GetMicroVM`, `ListMicroVMs`, and `ListMicroVMsStream`. The gRPC endpoint defaults to

`localhost:9090`; a grpc-gateway HTTP endpoint defaults to `localhost:8090`, with REST mappings for the first four methods (`POST /v1alpha1/microvm`, `DELETE /v1alpha1/microvm/{uid}`, `GET /v1alpha1/microvm/{uid}`, `GET /v1alpha1/microvm/{namespace}`).

MicroVMSpec: OCI Images As Disks

The `MicroVMSpec` protobuf message (package `flintlock.types`, proto3) captures the full declaration of a microVM:

```
string id = 1
string namespace = 2
map<string, string> labels = 3
int32 vcpu = 4
int32 memory_in_mb = 5
Kernel kernel = 6
optional Initrd initrd = 7
Volume root_volume = 8
repeated Volume additional_volumes = 9
repeated NetworkInterface interfaces = 10
map<string, string> metadata = 11
google.protobuf.Timestamp created_at = 12
google.protobuf.Timestamp updated_at = 13
google.protobuf.Timestamp deleted_at = 14
optional string uid = 15
```

The `Kernel` message (selected fields) carries `string image = 1`, `map<string,string> cmdline = 2`, `optional string filename = 3` (to name the kernel file within the image), and `bool add_network_config = 4`: the kernel is an OCI image reference, not a local path. `Volume.VolumeSource` has `optional string container_source = 1`, meaning disk images are also specified as OCI image references, pulled through containerd. The `NetworkInterface.IfaceType` enum has two values: `MACVTAP = 0` (required for Cloud Hypervisor) and `TAP = 1` (the only option Firecracker supports). `MicroVMStatus.MicroVMState` tracks `PENDING`, `CREATED`, `FAILED`, and `DELETING`.

Containerd As The Storage Layer

Flintlock uses containerd as its storage and pull infrastructure, connecting to the local containerd socket at `/run/containerd/containerd.sock`. It operates in the `"flintlock"` namespace — separate from the `"default"` namespace that other tools use — with volumes backed by the devmapper snapshotter (`ContainerdVolumeSnapshotter = "devmapper"`) and kernel and initrd images pulled using the native snapshotter (`ContainerdKernelSnapshotter = "native"`). State is rooted at `/var/lib/flintlock`; configuration lives in `/etc/opt/flintlockd`. The primary VMM backend is Firecracker via `firecracker-microvm/firecracker-go-sdk v1.0.0`; Cloud Hypervisor is experimentally supported, selected at startup with the `--default-provider` flag.

The pattern of block devices from devmapper snapshots being passed directly to Firecracker as virtio-blk drives is the same as in firecracker-containerd, but the purpose differs: firecracker-containerd passes a container's OCI layer as a block device to run an application inside the VM; flintlock passes an OS disk image as a block device to run a Kubernetes node on top of the VM.

The default resource limits, from `github.com/weaveworks/flintlock/pkg/defaults`: two vCPUs, 1024 MB of memory, a ten-second timeout on VM deletion, a ten-minute resync period, and a maximum of ten retries. The Firecracker binary is resolved from `$PATH` with `FirecrackerDetach = true`, meaning the VMM process is detached from the daemon's process group.

Cluster API Integration

The operational interface for flintlock at scale is `liquidmetal-dev/cluster-api-provider-microvm` (CAPMVM), a Cluster API Infrastructure Provider registered with `clusterctl` as `InfrastructureProvider` named `"microvm"`. Installation is `clusterctl init -i microvm`; CNI integration (for example, Cilium) requires the feature flag `EXP_CLUSTER_RESOURCE_SET=true`.

CAPMVM defines two core types. `MicrovmMachineSpec` embeds `microvm.VMSpec` inline and adds `SSHPublicKeys []microvm.SSHPublicKey` and `ProviderID *string`. `MicrovmMachineStatus` carries `Ready bool`, `VMState *microvm.VMState`, `Addresses []clusterv1.MachineAddress`, `FailureReason`, `FailureMessage`, and `Conditions` — the standard CAPI machine infrastructure-provider contract, with `GetConditions` / `SetConditions` and a `MachineFinalizer`. `MicrovmClusterSpec` fields include `ControlPlaneEndpoint clusterv1.APIEndpoint`, `SSHPublicKeys []microvm.SSHPublicKey`, `Placement`, `MicrovmProxy *flclient.Proxy`, and `TLSSecretRef string`. `MicrovmClusterStatus` carries `Ready bool`, `Conditions clusterv1.Conditions`, and `FailureDomains clusterv1.FailureDomains`.

A lighter-weight alternative exists for smaller deployments: the `liquidmetal-dev/microvm-operator`, a plain Kubernetes operator (not CAPI) for creating batches of microVMs on flintlock-managed devices. Its README marks it as proof of concept.

Where The Two Books Meet

The containerd book traces the container from a `containerd.Create` call down through the shim v2 protocol, the snapshotter, the CNI plugin chain, and the TC networking primitives. This book traces the microVM from a Firecracker API call down through KVM ioctls, virtio transports, and the hardware isolation boundary. The junction between the two traces is the containerd shim.

Topic	Containerd book	MicroVM book extension
Shim v2 / TaskService	`io.containerd.runc.v2` runs OCI containers	`aws.firecracker` / `io.containerd.kata.v2` substitute a VM for the container sandbox
devmapper snapshotter	Container layer storage on the host	Block device passed to Firecracker virtio-blk instead of mounted in the host filesystem
TAP + TC redirect	CNI networking for pods	Veth-to-TAP bridge for VM network interfaces in both firecracker-containerd and Kata
ttrpc	Containerd-to-shim control plane	Shim-to-in-VM-agent control plane (firecracker-containerd: port 10789; Kata: port 1024)

The hardware enforcement point is what the table cannot convey on its own. In both microVM paths, EPT or NPT in the CPU enforces the isolation boundary — a kernel vulnerability on the guest side cannot reach the host side without also defeating the hardware virtualization extension. That boundary is absent from plain container isolation, where the same kernel enforces security on both sides of every namespace wall. The shim is the hinge.

Chapters 4 through 8 of this book built the hardware side of that hinge: the VMCS, the `KVM_RUN` loop, the virtio queues, and the two-dimensional page tables. The chapters in the containerd book build the software side: the snapshotter API, the ttrpc shim protocol, the CNI invocation, and the event routing.

Sources And Further Reading

- firecracker-containerd architecture and shim design: https://github.com/firecracker-microvm/firecracker-containerd/blob/main/docs/architecture.md and https://github.com/firecracker-microvm/firecracker-containerd/blob/main/docs/shim-design.md
- firecracker-containerd `runtime/service.go` (VSOCK constants, drive mounting): https://github.com/firecracker-microvm/firecracker-containerd/blob/main/runtime/service.go
- firecracker-containerd snapshotter design: https://github.com/firecracker-microvm/firecracker-containerd/blob/main/docs/snapshotter.md
- firecracker-containerd drive stub pre-allocation: https://github.com/firecracker-microvm/firecracker-containerd/blob/main/docs/design-approaches.md
- firecracker-containerd networking (tc-redirect-tap): https://github.com/firecracker-microvm/firecracker-containerd/blob/main/docs/networking.md
- firecracker-containerd proto types: https://github.com/firecracker-microvm/firecracker-containerd/blob/main/proto/types.proto
- firecracker-containerd agent README: https://github.com/firecracker-microvm/firecracker-containerd/blob/main/agent/README.md

- Firecracker VSOCK multiplexing protocol: https://github.com/firecracker-microvm/firecracker/blob/main/docs/vsock.md
- Firecracker VSOCK internal implementation: https://github.com/firecracker-microvm/firecracker-containerd/blob/main/internal/vm/vsock.go
- containerd runtime v2 protocol: https://github.com/containerd/containerd/blob/main/docs/runtime-v2.md
- containerd TaskService proto: https://github.com/containerd/containerd/blob/main/api/runtime/task/v2/shim.proto
- kata-agent Rust source and README: https://github.com/kata-containers/kata-containers/blob/main/src/agent/README.md
- kata-agent AgentService proto (43 methods): https://github.com/kata-containers/kata-containers/blob/main/src/libs/protocols/protos/agent.proto
- Kata VSOCK design: https://github.com/kata-containers/kata-containers/blob/main/docs/design/VSocks.md
- Kata storage architecture (virtiofs, virtio-scsi, DAX): https://github.com/kata-containers/kata-containers/blob/main/docs/design/architecture/storage.md
- Kata Kubernetes integration (RuntimeClass, sandbox vs. container): https://github.com/kata-containers/kata-containers/blob/main/docs/design/architecture/kubernetes.md
- Kata networking (TC redirect, MACVTAP replacement): https://github.com/kata-containers/kata-containers/blob/main/docs/design/architecture/networking.md
- Kata hypervisor backends and version table: https://github.com/kata-containers/kata-containers/blob/main/docs/design/virtualization.md
- Kata on Firecracker (devmapper requirement, configuration-fc.toml): https://github.com/kata-containers/kata-containers/blob/main/docs/how-to/how-to-use-kata-containers-with-firecracker.md
- Kata 4.0 preview (runtime-rs default, Go runtime deprecation): https://katacontainers.io/blog/release-4-0-0-preview/
- flintlock repository: https://github.com/liquidmetal-dev/flintlock
- flintlock MicroVM gRPC service proto: https://github.com/liquidmetal-dev/flintlock/blob/main/api/services/microvm/v1alpha1/microvms.proto
- flintlock MicroVMSpec proto: https://buf.build/weaveworks-liquidmetal/flintlock/raw/58fe9b39fd874ce7abc0773abe71f072/-/types/microvm.proto
- flintlock default constants: https://pkg.go.dev/github.com/weaveworks/flintlock/pkg/defaults
- flintlock go.mod (SDK versions): https://github.com/liquidmetal-dev/flintlock/blob/main/go.mod
- Cluster API provider for microVMs (CAPMVM): https://github.com/liquidmetal-dev/cluster-api-provider-microvm
- CAPMVM MicrovmMachine types: https://github.com/liquidmetal-dev/cluster-api-provider-microvm/blob/main/api/v1alpha1/microvmmachine_types.go

- CAPMVM MicrovmCluster types: https://github.com/liquidmetal-dev/cluster-api-provider-microvm/blob/main/api/v1alpha1/microvmcluster_types.go

Chapter 23: The VMM Landscape

Firecracker's device list fits on a napkin: `virtio-net`, `virtio-blk`, `virtio-vsock`, `virtio-balloon`, a 16550A UART, and an i8042 stub. Six devices, no PCI bus, no display, no audio, no ISA timer chain beyond what the interrupt controller requires. That restraint is not an accident of being new — it is the load-bearing design decision the whole microVM argument rests on. But to appreciate what was removed, you have to see what was not.

Three VMMs define the space around Firecracker: **QEMU**, the full-featured baseline that can emulate any machine ever sold; **crosvm**, the Google ChromeOS VMM from which Firecracker's codebase was directly forked; and **Cloud Hypervisor**, the modern Rust sibling targeting general-purpose cloud workloads. None of these is better than the others in any absolute sense. Each embodies a specific tradeoff between feature breadth and attack surface, and understanding that tradeoff is what makes Firecracker's six devices legible as engineering rather than as minimalism for its own sake.

The Shared Substrate

All three Rust VMMs share code through the **rust-vmm** umbrella project, and that common ancestry shapes what is comparable between them.

rust-vmm was founded in December 2018 when engineers from Amazon, Google, Intel, and Red Hat began extracting core virtualization code from crosvm and Firecracker into shared crates. Firecracker itself had already been forked from crosvm's codebase before Amazon open-sourced it on November 27, 2018. The first shared artifact was the `vm-memory` crate, whose initial commit (`83f61119` on GitHub) was derived simultaneously from crosvm commit `186eb8b0` and Firecracker commit `80128ea6`. Today rust-vmm hosts more than 30 crates; current contributors include Alibaba, AWS, Intel, Google, Linaro, and Red Hat.

The crates that appear in VMM comparisons throughout this chapter are: `kvm-ioctls` (v0.25.0), which provides safe Rust wrappers over the three-level KVM file descriptor API; `kvm-bindings` (v0.14.1), the FFI layer beneath it; `vm-memory` (v0.17.1), which decouples memory consumers from memory providers through `GuestMemoryMmap` and `GuestAddress`; `linux-loader` (v0.13.2) for loading ELF, bzImage, and PE kernels into guest memory; `seccompiler` (v0.5.0) for compiling and installing per-thread BPF seccomp filters; and `virtio-queue` (v0.17.0), the split-ring virtqueue implementation that every virtio device in the Rust VMMs drives. Cloud Hypervisor's `Cargo.toml` workspace manifest pins all of these versions explicitly. Firecracker uses a subset. crosvm, as of the last documentation published on `chromium.googlesource.com`, had not formally consumed rust-vmm crates in production, citing friction around non-Linux OS support for `kvm-bindings` and `vmm-sys-util` — though that document dates to late 2020 and may no longer reflect the current codebase.

The shared substrate means that a virtqueue, a guest memory region, or a KVM ioctl wrapper works the same way across all three Rust VMMs. The differences lie above the crate boundary: which devices each VMM implements, how it isolates them, and what workloads it is designed to serve.

QEMU: The Full-Featured Baseline

Fabrice Bellard published QEMU's first preview in 2003. It is a hosted hypervisor and machine emulator written in C, licensed under GPL-2.0-only for its core, and it remains the single tool in this space that can boot a guest of any ISA on a host of any other ISA without hardware assistance. That capability explains everything that follows.

QEMU runs in two fundamentally different modes. In **TCG mode** (Tiny Code Generator), a software JIT cross-compiles guest basic blocks to host instructions at runtime, with no kernel support required. This is how QEMU can emulate a MIPS board on an x86-64 laptop. In **KVM mode**, selected with `-accel kvm` or `-enable-kvm`, QEMU delegates vCPU execution to the kernel's `kvm.ko` module and achieves near-native speed; TCG still handles device model emulation and any guest code that causes a VM exit. Other acceleration backends exist — Hypervisor.Framework on macOS (`-accel hvf`), WHPX on Windows, MSHV for Hyper-V — but KVM is the standard production path on Linux.

The device model is where QEMU's scope becomes concrete. The `hw/` directory in the QEMU source tree contains approximately 70 subdirectories covering device families: `acpi`, `block`, `char`, `display`, `dma`, `i386`, `ide`, `input`, `net`, `nvme`, `pci`, `scsi`, `timer`, `tpm`, `usb`, `vfio`, `virtio`, and dozens more from `9pfs` to `xen` and `xtensa`. QEMU's own security documentation identifies the primary attack surface as "emulated devices" and lists as untrusted inputs: guest code, VNC and SPICE connections, NBD and live-migration network protocols, user-supplied disk images, device trees, and PCI and USB passthrough devices. The breadth of that list is not incidental — it follows directly from the breadth of what QEMU emulates.

Even when KVM is doing the vCPU work, a QEMU process in default x86-64 mode emulates a full PCI bus hierarchy, ISA devices (the i8254 PIT, i8257 DMA controller, CMOS/RTC, i8259 PIC), USB host controllers, VGA and display adapters, sound cards, and the BIOS or OVMF firmware path. Each emulated device is code reachable from the guest over a hardware interface. Every byte the guest can write to a device register is a potential attack surface. This is not a critique of QEMU — it is the inevitable cost of universal compatibility.

QEMU's security policy at `qemu.org/docs/master/system/security.html` describes isolation options: SELinux, AppArmor, resource limits, cgroups, Linux namespaces, and seccomp via `--sandbox`. The key word is "options": seccomp is not on by default. Hardening QEMU for multi-tenant use requires explicit, per-deployment configuration; the tool does not arrive hardened.

The microvm Machine Type

QEMU does contain one concession to the minimalist camp: a machine type called `microvm`, selected with `-machine microvm`. This machine type strips away PCI, ACPI, and most legacy hardware, exposing up to 8 virtio-MMIO devices, one optional ISA serial port, LAPIC, IOAPIC, kvmclock, and `fw_cfg`. It does not support device hotplug or live migration across QEMU versions.

The `microvm` machine type is useful as a side-by-side comparison with Firecracker: same virtio-MMIO transport, similarly stripped device set. The difference is that it still runs inside QEMU's full codebase — the TCG JIT, the multi-ISA emulation engine, all 70 device family subdirectories — even when none of them are exposed to a given guest. The attack surface of the process includes code that is present but not reachable through the current machine type's configuration. Firecracker's device count is low because the code for everything else was never written, not because it is compiled out.

Intel recognized this gap. Its **NEMU** project (`github.com/intel/nemu`, archived April 14, 2021) was an attempt to strip QEMU specifically for cloud workloads. The archived repository's own README now redirects visitors: "Cloud Hypervisor is the successor." That sentence closes one lineage and opens the next.

Cloud Hypervisor: The Modern rust-vmm Sibling

Cloud Hypervisor grew from NEMU's successor effort and was relaunched under the `cloud-hypervisor` GitHub organization at v0.4.0. It is now governed by the Linux Foundation as "a Series of LF Projects, LLC." Its `README.md` states plainly that "a large part of the Cloud Hypervisor code is based on either the Firecracker or the crosvm project's implementations." Supporting organizations include Alibaba, AMD, Ampere, ARM, ByteDance, Cyberus Technology, Intel, Microsoft, SAP, and Tencent Cloud. The current release as of this writing is **v52.0**, released May 14, 2026, following an approximately monthly cadence that has been in place since v15.0.

Two Hypervisor Backends

Cloud Hypervisor runs on two backends compiled into a single binary with run-time detection, a capability introduced at v26.0. The primary backend is **KVM** on Linux. The second is **MSHV** — the Microsoft Hypervisor interface, used on Azure hosts and Windows Hyper-V. This dual-backend binary stands in contrast to Firecracker, which is KVM-only, and to crosvm, which supports KVM plus Gunyah, GenieZone, and Halla for Android hardware, plus WHPX and HAXM on Windows.

The minimum recommended host kernel for Cloud Hypervisor is 5.13 for required KVM functionality; CI runs against 5.15. Supported architectures are x86-64 (primary), AArch64 (primary, requiring GICv3), and riscv64 (experimental). Supported guest operating systems are 64-bit Linux and Windows 10 / Windows Server 2019 — the Windows guest support alone distinguishes Cloud Hypervisor from both Firecracker and crosvm in a meaningful way for enterprise workloads.

The Device Model

The most structurally important decision in Cloud Hypervisor's device model is transport: all virtio devices use **virtio-PCI** exclusively. virtio-MMIO was removed at v0.11.0 to simplify the code and reduce the testing burden. Firecracker uses virtio-MMIO throughout. This single decision means that Cloud Hypervisor requires a guest kernel with PCI support and a host that can set up a PCI bus in the VM, whereas Firecracker's guests need no PCI driver at all.

The built-in virtio device set, as of v52.0, spans ten devices: `virtio-blk` (default `io_uring` backend since v0.11.0), `virtio-console`, `virtio-net` (with multi-queue and multi-thread since v0.5.0 and rate limiting), `virtio-pmem`, `virtio-rng`, `virtio-vsock` (forked from Firecracker's vsock implementation, as acknowledged in `docs/device_model.md`), `virtio-iommu`, `virtio-mem` (the virtio 1.2 memory device for hotplug, since v0.7.0), `virtio-balloon` (with free-page reporting since v22.0), and `virtio-watchdog` (experimental, since v0.11.0).

Beyond the built-in virtio devices, Cloud Hypervisor supports four **vhost-user offload backends**: `vhost-user-blk` for high-performance block via SPDK, `vhost-user-net` for DPDK-backed networking, `vhost-user-fs` for shared filesystems via `virtiofsd`, and `vhost-user-generic` (added in v52.0), which allows arbitrary backends without requiring the VMM to know the device type. Each of these runs in a separate process; the VMM and the backend communicate over a UNIX socket.

Emulated legacy devices are kept deliberately narrow: a 16550A serial port on x86-64 (PL011 UART on AArch64), RTC/CMOS, I/O APIC, i8042, ARM PL061 GPIO (for AArch64 shutdown), and an ACPI device as the default shutdown and reboot path. No emulated e1000, no IDE controller, no PS/2 bus, no i8254 PIT by default. This is the "wider than Firecracker, narrower than QEMU" line Cloud Hypervisor draws.

VFIO passthrough of physical PCI and PCIe devices has been available since v0.1.0, with hotplug since v0.6.0. Version 52.0 added support for the modern `iommufd` / `vfio-cdev` interface introduced in Linux 6.6, enabling selective BAR mapping, sub-page BAR expansion, MSI-X synchronization, and lazy GSI allocation.

Memory and CPU Architecture

Cloud Hypervisor's memory configuration is richer than Firecracker's precisely because its target workload is longer-lived and more varied. The `--memory` flag accepts fields including `size`, `mergeable` (KSM), `hugepages`, `prefault` (using `MAP_POPULATE`), `shared` (using `mmap(MAP_SHARED)`), `hotplug_method` (either `acpi` or `virtio-mem`), and `hotplug_size`. ACPI hotplug increments must be multiples of 128 MiB; `virtio-mem` carries no such alignment constraint. Multiple PCIe segments are available since v20.0, configured via `--platform num_pci_segments=<N>,iommu_segments=<range>`.

The `virtio-iommu` device provides a paravirtualized IOMMU that eliminates shadow page table complexity. When a physical IOMMU and a VFIO device are both present, DMA remapping tables are updated via VFIO whenever the guest updates its mappings, enabling nested VFIO passthrough; hugepages reduce IOMMU mapping overhead substantially in this configuration.

vCPU topology follows a four-level model: `threads:cores:dies:packages`, defaulting to `1:1:1:1`. vCPU hotplug is supported by configuring `max` greater than `boot` at startup and then onlining CPUs in the guest via `/sys/devices/system/cpu/cpu*/online`. AMX support for x86 was added at v23.0. SMT side-channel mitigation is handled with the `core_scheduling` option (added v52.0), which supports modes `vm`, `vcpu`, and `off`.

Version 52.0 also added KVM SEV-SNP support for confidential VMs, using `KVM_CREATE_GUEST_MEMFD` for private guest memory. Firmware is packaged as IGVM; the kernel, command line, and initrd are included in the launch measurement.

Live Migration and Snapshot

Cloud Hypervisor is designed for the full lifecycle of a cloud VM, which includes moving it. Live migration uses a UNIX socket for local transfer or TCP for remote; TLS and mTLS are available on the TCP path. Version 52.0 added multi-connection TCP (1 to 128 parallel connections) to saturate high-bandwidth links. Migration supports both precopy (the default) and postcopy modes. Protocol versioning is strict: each release sends its current version number and accepts the immediately preceding version; jumping multiple releases requires stepping through intermediate versions. Default maximum downtime is 300 ms; timeout is 3600 s.

Snapshots are written to a directory containing three files: `config.json` (the full VM configuration, human-readable), `memory-ranges` (raw guest RAM), and `state.json` (per-component state). The `ondemand` restore mode, added in v52.0, uses `userfaultfd` to fault pages in lazily, reducing time-to-first-instruction on restore. VFIO devices are excluded from snapshot and restore.

None of Firecracker's published specifications mention live migration. The workloads Firecracker serves — serverless function invocations lasting tens to hundreds of milliseconds — do not need it. The workloads Cloud Hypervisor serves — persistent cloud VMs, containers-as-VMs, Windows guests, confidential compute instances — do.

---

crosvm: The ChromeOS Ancestor

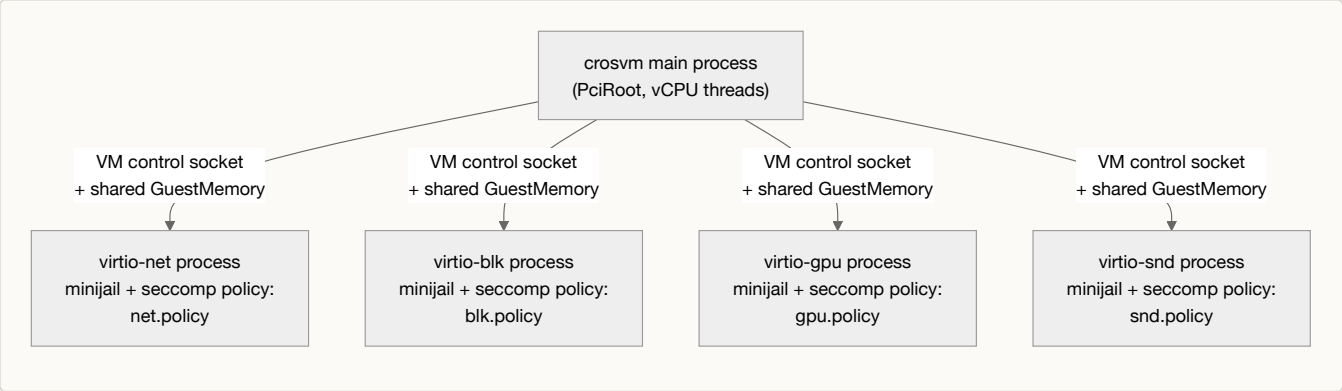
Google built crosvm for ChromeOS's Crostini Linux container runtime and for the Android guest (ARCVM). crosvm's README states that Firecracker "used [crosvm] as the basis for their own VMM," and the history of the `vm-memory` crate confirms it: the first rust-vmm artifact was derived from both codebases simultaneously in December 2018. crosvm has since expanded to Android's TerminalApp, Cuttlefish (Google's virtual Android device platform), and Windows hosts.

Process-Per-Device Isolation

crosvm's defining architectural decision — the one that distinguishes its security model most sharply from Firecracker's — is its **process-per-device sandbox model**. Each virtio device backend runs in a sandboxed child process, not as a thread within the main VMM process. The main VMM forks and jails

each device process using **minijail**, a Google library that wraps Linux namespaces and seccomp-BPF.

Each jailed device process receives three layers of containment: VFS, PID, user, and network namespaces via `pivot_root`; a per-device-type BPF seccomp policy from `jail/seccomp/{arch}/{device}.policy`; and Linux capability dropping. The main process retains `PciRoot` coordination; jailed device processes communicate back via VM control sockets over shared `GuestMemory`.



The blast-radius argument here is different from Firecracker's. In Firecracker, a compromised device path is contained by a per-thread seccomp-BPF filter inside a single monolithic process — the attacker is in the same process but limited in which syscalls they can make. In crosvm, a compromised `virtio-net` backend is confined to a separate process with its own namespace and its own seccomp policy; to affect the VMM main process, an attacker must additionally escape the process boundary. The tradeoff is that process-per-device adds IPC overhead and complexity in the main process's `PciRoot` coordination, costs that Firecracker avoids by accepting the monolithic model.

Hypervisor Backends

crosvm runs on more hypervisor backends than either Firecracker or Cloud Hypervisor: **KVM** on Linux (primary), **Gunyah**, **GenieZone**, and **Halla** on Linux and Android hardware, plus **WHPX** and **HAXM** on Windows. This breadth reflects crosvm's deployment reality — Android hardware may run non-KVM hypervisors, and the same crosvm codebase must function across all of them.

Device Set

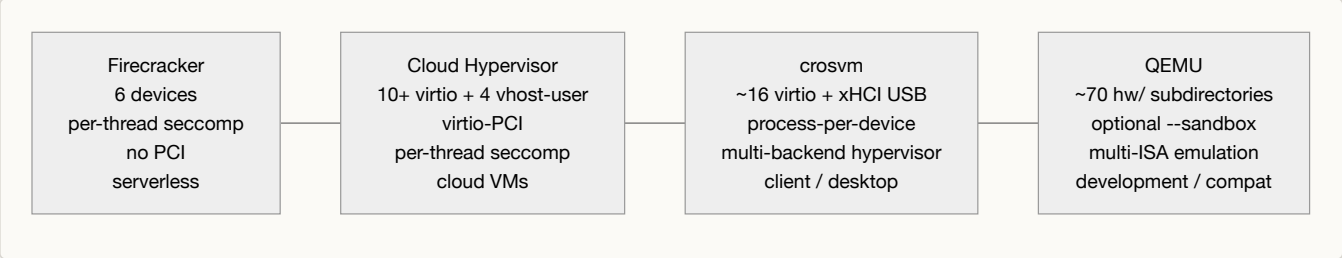
crosvm's device scope follows its workload: a ChromeOS or Android desktop environment, not a serverless function. The virtio device list includes `virtio-blk` (raw, QCOW2, zstd, and Android sparse image formats), `virtio-net` (vhost and slirp backends), `virtio-vsock`, `virtio-gpu` (2D, virglrenderer 3D, gfxstream, and Vulkan), `virtio-snd` (CRAS and AAudio backends), `virtio-fs` (FUSE), `virtio-9p`, `virtio-input`, `virtio-balloon`, `virtio-console`, `virtio-rng`, `virtio-iommu`, `virtio-tpm`, `virtio-pmem`, and experimental `virtio-video` and `virtio-scsi`. Emulated legacy devices include CMOS/RTC, i8042, serial (I/O port), and xHCI USB passthrough.

Wayland display forwarding operates through `virtio-gpu` cross-domain mode and requires a guest Linux kernel at version 5.16 or later with `CONFIG_DRM_VIRTIO_GPU` enabled.

The devices Firecracker deliberately excludes — `virtio-gpu`, audio, 9P filesystem sharing, USB passthrough, and display forwarding — are exactly the devices crosvm includes as first-class features. Neither list is wrong; they reflect genuinely different user-visible workloads.

Scope, Attack Surface, and Intended Workload

The four VMMs arrange into a clear gradient when you hold them against the same axes: how many device types each exposes, how it isolates them, and what workload justifies the design.



Firecracker draws the smallest perimeter: six devices confirmed in the FAQ and design document, no PCI bus, no GPU, no audio, no display. The measurable results are stated in `SPECIFICATION.md` and enforced in CI: VMM startup to API socket in at most 8 CPU milliseconds, guest `/sbin/init` from `InstanceStart` in at most 125 milliseconds, and no more than 5 MiB of VMM memory overhead per microVM at 1 vCPU and 128 MiB of guest RAM. Guest CPU performance stays above 95 percent of bare metal; network throughput reaches 14.5 Gbps at 80 percent host CPU utilization. Max vCPUs per microVM is 32. The narrow device list is what makes those bounds achievable and verifiable in CI.

The seccomp posture is the other half of Firecracker's security argument. The three thread types — the API thread, the VMM thread, and one vCPU thread per guest CPU — each run under a separate BPF filter, installed before any guest code executes. The API thread's filter allows exactly `FIONBIO` among ioctls. Selected ioctls the VMM thread's filter allows include `KVM_SET_USER_MEMORY_REGION`, `KVM_IOEVENTFD`, `KVM_IRQFD`, `TUNSETIFF`, `TUNSETOFFLOAD`, `TUNSETVNETHDRSZ`, `KVM_GET_DIRTY_LOG`, `KVM_GET_IRQCHIP`, `KVM_GET_CLOCK`, and `KVM_GET_PIT2` (the last three enable snapshot and restore of interrupt-controller and timer state). The vCPU thread's filter allows `KVM_RUN` and GET ioctls for registers and CPU state; the full enumeration is in the seccomp JSON files rather than reproduced here, because the set changes across releases. These filters are compiled into the binary for exactly two target triples: `x86_64-unknown-linux-musl` and `aarch64-unknown-linux-musl`. The files live at `resources/seccomp/x86_64-unknown-linux-musl.json` and `resources/seccomp/aarch64-unknown-linux-musl.json`.

Cloud Hypervisor takes a wider stance at every axis. Ten built-in virtio devices, four vhost-user offload backends, VFIO passthrough, Windows guest support, live migration, vCPU hotplug, memory hotplug, and confidential VM support via KVM SEV-SNP. Its seccomp posture uses the `seccompiler` rust-vmm

crate for per-thread filters, but the set of allowed operations is necessarily larger because the device model is larger. The workloads that justify the additional surface area — Windows guests, persistent cloud VMs, VFIO-attached hardware, confidential compute — cannot fit inside Firecracker's perimeter.

crosvm trades off differently again. Its device set is the widest of the three Rust VMMs — `virtio-gpu` with Vulkan, `virtio-snd`, Wayland forwarding, xHCI USB, 9P and virtio-fs — but it contains the blast radius of any single compromised device through process isolation rather than through device omission. The seccomp policies at `jail/seccomp/{arch}/{device}.policy` are per-device-type rather than per-thread, because each device lives in its own process. A compromised `virtio-gpu` backend is confined to the GPU policy and the GPU process's namespaces; it does not directly threaten the VMM main process. crosvm's threat model is "the guest is interactive and relatively trusted; the device boundary is where you isolate against bugs" — a different assumption from Firecracker's "the guest is hostile and the perimeter is the whole VMM."

QEMU maximizes compatibility. The `hw/` subdirectory count of approximately 70 is the structural proxy for attack surface: each subdirectory is a family of emulated hardware, each family is code reachable from the guest, and the guest in QEMU's default configuration can reach most of them. Hardening requires explicit opt-in — `--sandbox on`, an AppArmor profile, cgroup limits — and even with all of these applied, the underlying device model is present in the process. QEMU is the right tool for development, cross-ISA testing, CI environments, and any workload that requires hardware it cannot buy; it is the wrong starting point for a multi-tenant production deployment of untrusted code without a significant hardening investment.

The KVM Interface Each VMM Uses

All three Rust VMMs reach the kernel through `kvm-ioctls` v0.25.0; QEMU uses its own C wrappers. The KVM API itself is the same three-level file descriptor interface for all of them: a `/dev/kvm` fd for system-level operations, a VM fd opened with `KVM_CREATE_VM`, and per-vCPU fds opened with `KVM_CREATE_VCPU`. `VmFd` in `kvm-ioctls` exposes `KVM_SET_USER_MEMORY_REGION`, `KVM_CREATE_IRQCHIP`, `KVM_IOEVENTFD`, `KVM_IRQFD`, `KVM_GET_DIRTY_LOG`, and `KVM_CREATE_GUEST_MEMFD` (for confidential VM private memory), among others. `VcpuFd` exposes `KVM_RUN`, `KVM_GET_REGS`, `KVM_SET_REGS`, `KVM_GET_SREGS`, `KVM_SET_CPUID2`, `KVM_GET_LAPIC`, `KVM_SET_LAPIC`, `KVM_GET_MSRS`, `KVM_SET_MSRS`, `KVM_GET_XSAVE`, `KVM_GET_NESTED_STATE`, and more.

> ***Note:*** *Opening `/dev/kvm` requires membership in the `kvm` group on most Linux distributions, or root access. Any process that holds a VM fd or a vCPU fd has host-kernel access at the level of those handles. Run VMMs as non-root, apply seccomp, and restrict `/dev/kvm` permissions before exposing any VMM to untrusted guest workloads.*

Each VMM uses a subset of the available ioctls. Firecracker's seccomp filter is the most explicit statement of which subset: it is the list of ioctls the process is allowed to call at all, enforced by the kernel's BPF machinery, not merely by the code paths the VMM happens to exercise. Cloud Hypervisor and crosvm do not publish an equivalent enumeration in a single file; their allowed sets are wider because their device models demand it.

A Note on the virtio 1.2 Registry

The OASIS virtio 1.2 specification defines 19 device types by ID. Comparing each VMM against that registry makes the gaps visible at a glance. Firecracker implements IDs 1 (Network), 2 (Block), 5 (Balloon, since v0.24.0), and 19 (vsock); Cloud Hypervisor adds IDs 3 (Console / virtio-console), 4 (RNG), 23 (IOMMU), 24 (Memory / virtio-mem), 26 (File System, via vhost-user-fs), and 27 (PMEM); crosvm adds IDs 16 (GPU), 18 (Input), 25 (Sound), 26 (File System), and more. IDs 8 (SCSI), 20 (Crypto), and 29 (Administration) appear in none of the three Rust VMMs' stable device sets as of this writing.

The spec registry is useful not as a scorecard but as a stable coordinate system: when a new device type appears in a VMM, you can look up its ID and description in the OASIS document rather than treating it as a proprietary feature.

The next chapter turns back to Firecracker specifically and examines how its seccomp filters are constructed — the mechanism that makes the thread model's security promises concrete.

Sources And Further Reading

- rust-vmm community and crate inventory: https://github.com/rust-vmm/community
- vm-memory initial commit (crosvm + Firecracker lineage): https://github.com/rust-vmm/vm-memory/commit/83f61119faed409f4569d2418fd39f8769b919fc
- crosvm README (hypervisor backends, origin, device set): https://chromium.googlesource.com/chromiumos/platform/crosvm/+/HEAD/README.md
- crosvm rust-vmm integration document: https://chromium.googlesource.com/chromiumos/platform/crosvm/+/master/docs/rust-vmm.md
- crosvm architecture overview (process-per-device model): https://crosvm.dev/book/architecture/overview.html
- crosvm device list: https://crosvm.dev/book/devices/index.html
- crosvm Wayland forwarding (kernel 5.16 requirement): https://crosvm.dev/book/devices/wayland.html
- Firecracker FAQ (6 devices, memory footprint, boot latency): https://github.com/firecracker-microvm/firecracker/blob/main/FAQ.md

- Firecracker design document (thread model, max 32 vCPUs, device set, threat model): https://github.com/firecracker-microvm/firecracker/blob/main/docs/design.md
- Firecracker SPECIFICATION.md (CI-enforced performance bounds): https://raw.githubusercontent.com/firecracker-microvm/firecracker/main/SPECIFICATION.md
- Firecracker seccomp allowlists (x86-64, ioctl-level): https://raw.githubusercontent.com/firecracker-microvm/firecracker/main/resources/seccomp/x86_64-unknown-linux-musl.json
- Firecracker seccomp file tree (two target triples): https://github.com/firecracker-microvm/firecracker/tree/main/resources/seccomp
- Cloud Hypervisor repository root: https://github.com/cloud-hypervisor/cloud-hypervisor
- Cloud Hypervisor v52.0 release notes: https://github.com/cloud-hypervisor/cloud-hypervisor/releases/tag/v52.0
- Cloud Hypervisor README (governance, backends, architectures, minimum host kernel): https://github.com/cloud-hypervisor/cloud-hypervisor/blob/main/README.md
- Cloud Hypervisor workspace manifest (rust-vmm version pins): https://github.com/cloud-hypervisor/cloud-hypervisor/blob/main/Cargo.toml
- Cloud Hypervisor device model (virtio-PCI-only, vhost-user, emulated legacy): https://github.com/cloud-hypervisor/cloud-hypervisor/blob/main/docs/device_model.md
- Cloud Hypervisor hotplug (ACPI GED, vCPU hotplug, 128 MiB ACPI constraint): https://github.com/cloud-hypervisor/cloud-hypervisor/blob/main/docs/hotplug.md
- Cloud Hypervisor IOMMU (virtio-iommu, nested VFIO passthrough, hugepages): https://github.com/cloud-hypervisor/cloud-hypervisor/blob/main/docs/iommu.md
- Cloud Hypervisor live migration (transport, protocol versioning, downtime): https://github.com/cloud-hypervisor/cloud-hypervisor/blob/main/docs/live_migration.md
- Cloud Hypervisor snapshot format (config.json, memory-ranges, state.json): https://github.com/cloud-hypervisor/cloud-hypervisor/blob/main/docs/snapshot_restore.md
- Cloud Hypervisor CPU model (topology, AMX, SMT core scheduling): https://github.com/cloud-hypervisor/cloud-hypervisor/blob/main/docs/cpu.md
- Cloud Hypervisor memory configuration fields: https://intelkevinputnam.github.io/cloud-hypervisor-docs-HTML/docs/memory.html
- kvm-ioctls v0.25.0 VmFd method list: https://docs.rs/kvm-ioctls/0.25.0/kvm_ioctls/struct.VmFd.html
- kvm-ioctls v0.25.0 VcpuFd method list: https://docs.rs/kvm-ioctls/0.25.0/kvm_ioctls/struct.VcpuFd.html
- rust-vmm vhost-device backends: https://github.com/rust-vmm/vhost-device
- Intel NEMU archived repository (Cloud Hypervisor predecessor): https://github.com/intel/nemu
- QEMU security documentation (attack surface, isolation options): https://www.qemu.org/docs/master/system/security.html
- QEMU microvm machine type: https://www.qemu.org/docs/master/system/i386/microvm.html

- QEMU hw/ device directory tree: https://github.com/qemu/qemu/tree/master/hw
- OASIS virtio 1.2 specification and device type registry: https://docs.oasis-open.org/virtio/virtio/v1.2/virtio-v1.2.html
- Cloud Hypervisor project site (supporting organizations): https://www.cloudhypervisor.org/
- rust-vmm founding history (December 2018): https://opensource.com/article/19/3/rust-virtual-machine