The Great Escape: Why Containers Won’t Save You

(and How MicroVMs Will)

Part 3: The Abyss

Laurent Bobelin

laurent.bobelin@insa-cvl.fr

INSA
INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
CENTRE-VAL DE LOIRE

ToC

1. The Great Escape

2. Hardware Assisted Virtualization
2.1 VM Environment
2.2 Hardware Security of System Calls
2.3 Hardware and Hypervisors

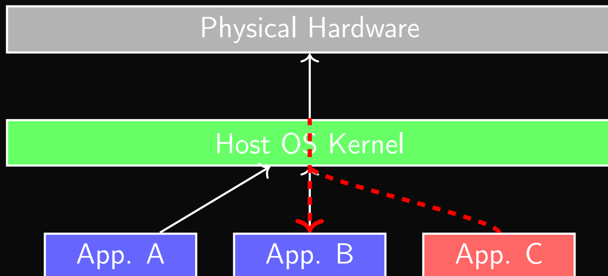
3. MicroVM and OCI: The Great Jail
3.1 Overview
3.2 Kata-Containers
3.3 Firecracker

4. Conclusion

The Container Security Problem

Traditional Container Architecture

In a standard setup, all containers run as isolated processes **on the same host kernel**.



The Risk

A single kernel vulnerability exploited by C can be used to attack the host and all other containers. **The isolation boundary is too thin.**

Escaping from a Container is Still an Issue

Even with all the security deployed with *traditional* containers, escaping from it will still potentially:

- Lateral movement
- Compromise other containers
- Use of host resources
- ...

So the questions are:

- How may we avoid this?
- Is there a way to use more secure solutions? VM are supposed to be more secure BTW, but how?
- Do OCI Specifications may be of any help?

ToC

1. The Great Escape

2. Hardware Assisted Virtualization
2.1 VM Environment
2.2 Hardware Security of System Calls
2.3 Hardware and Hypervisors

3. MicroVM and OCI: The Great Jail
3.1 Overview
3.2 Kata-Containers
3.3 Firecracker

4. Conclusion

ToC

1. The Great Escape

2. Hardware Assisted Virtualization
2.1 VM Environment
2.2 Hardware Security of System Calls
2.3 Hardware and Hypervisors

3. MicroVM and OCI: The Great Jail
3.1 Overview
3.2 Kata-Containers
3.3 Firecracker

4. Conclusion

VM ecosystem: Hypervisor/VMM

Piece of software, firmware, or hardware that creates and runs **Virtual Machines (VMs)**.

Roles

- **abstracts** the physical hardware (CPU, Memory, Storage, Network) from the Operating System (OS).
- Each VM runs an independent OS, called the **Guest OS**.

Key Goals

- **Isolation**: Ensures that one VM's operation doesn't affect another or the host.
- **Sharing**: Allows multiple VMs to safely and efficiently share the same underlying physical resources.

KVM and QEMU

KVM (Kernel-based Virtual Machine)

- Kernel Module (`kvm.ko`).
- Host Linux OS → **hypervisor**.
- Provides **Hardware-Assisted Virtualization** (HVA) by using CPU extensions (Intel VT-x, AMD-V).

QEMU (Quick Emulator)

- User-Space application
- For **Full System Emulation**.
- Emulates all necessary virtual hardware components (disk controller, network card, GPU, BIOS).

ToC

1. The Great Escape

2. Hardware Assisted Virtualization
2.1 VM Environment
2.2 Hardware Security of System Calls
2.3 Hardware and Hypervisors

3. MicroVM and OCI: The Great Jail
3.1 Overview
3.2 Kata-Containers
3.3 Firecracker

4. Conclusion

User Space and Kernel Space

- Modern operating systems separate execution into two domains:
  - **User space**: where applications run
  - **Kernel space**: where the operating system runs
- Applications in user space **cannot directly access**:
  - hardware devices
  - kernel memory
  - privileged CPU instructions
- This separation improves:
  - **security** (applications cannot corrupt the kernel)
  - **stability** (a crash in a program does not crash the system)
- But applications still need OS services:
  - file access
  - networking
  - process creation

System Calls

- **System calls** provide the controlled interface between **user space** and **kernel space**.
- Instead of accessing hardware directly, applications request services from the kernel.
- Typical system calls include:
  - `open()`, `read()`, `write()` (file access)
  - `fork()` / `exec()` (process management)
  - `socket()` (network communication)
- A system call triggers a **controlled transition** from user mode to kernel mode.
- This requires **hardware support** to enforce privilege levels and prevent applications from executing privileged operations directly.

How does the CPU enforce these privilege levels?

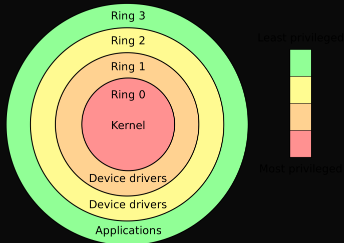
Protection Rings for x86

- Mechanism used by the CPU to enforce **fault isolation** and **security policies** between software components.
- Prevent low-privilege software (e.g., user applications) from corrupting or interfering with high-privilege software (e.g., the OS kernel).

In practice, 2 are used:

- ring 0 for the operating system,
- ring 3 for user software.

Protection Mechanism:
Attempt by Ring 3 code to execute a privileged instruction $\rightarrow$ CPU exception, $\rightarrow$ back to the secure Ring 0 kernel.



2. The Core Rings: Ring 0 and Ring 3

Ring 0: The Kernel (Most Privileged)

- **Access:** direct and unrestricted access to the CPU, hardware, memory, and I/O devices.
- **Code Running:** The Operating System **Kernel** (e.g., Linux, Windows NT core).
- **Operations:** Executes privileged instructions (halt, disabling interrupts, ...)

Ring 3: User Mode (Least Privileged)

- **Access:** Limited to its own process memory space; cannot directly access hardware.
- **Code Running:** All **User Applications** (browsers, word processors, games).
- **Operations:** Must request services from Ring 0 via a System Call (e.g., ‘syscall‘).

Hardware assistance for virtualization: allowing the same type of protection to be implemented for guest systems.

ToC

1. The Great Escape

2. Hardware Assisted Virtualization
2.1 VM Environment
2.2 Hardware Security of System Calls
2.3 Hardware and Hypervisors

3. MicroVM and OCI: The Great Jail
3.1 Overview
3.2 Kata-Containers
3.3 Firecracker

4. Conclusion

Hardware-Assisted Virtualization

Idea: add a ring -1 to the existing rings (AMD) or a root or non-root mode (Intel) designed for the hypervisor:

- VT-x (VMX) is the Intel technology
- AMD-V (ring -1) is AMD's.
- It amounts to roughly the same thing

Both manufacturers began distributing processors with these technologies in 2006.

VT-x: VMX

Enable/disable VT-x == VMX mode on/off:

- Non-VMX mode: nothing special
- VMX mode:
  - hypervisor: **VMX root** operation mode
  - Guest OS and app: **VMX non-root** operation mode
  - In both cases, the 4 levels (0,1,2,3) are present

VMX root mode == non-VMX root mode + special instructions for virtualization:

- vmxon and vmxoff to enter/exit VMX mode
- and vmlaunch and vmresume for VMs
- ...

Hardware manages VMCS (Virtual Machine Control Structure) with the state of registers for each VM:

- the guest OS works on registers natively
- The hardware saves the registers at each context switch.

IOMMU: Protecting Memory from Devices

- The CPU is not the only one that accesses memory:
  - peripherals can use **DMA** (Direct Memory Access)
  - e.g. NICs, NVMe controllers, GPUs
- Problem:
  - without extra protection, a device may read or write arbitrary physical memory through DMA
  - this can bypass normal CPU privilege checks
- The **IOMMU** (I/O Memory Management Unit):
  - translates device-visible I/O virtual addresses
  - enforces which memory regions a device is allowed to access
  - isolates devices from the rest of the system
- Intuition:
  - MMU $\Rightarrow$ CPU memory accesses
  - IOMMU $\Rightarrow$ device DMA accesses

IOMMU and Interrupt Remapping (IIRC)

- Devices interact with the system in two ways:
  1. **DMA** (reading/writing memory)
  2. **Interrupts** (signaling events to the CPU)

- The IOMMU provides two key protections:
  - **DMA remapping**:
    - controls which memory a device can access
  - **Interrupt remapping (IIRC)**:
    - controls how device interrupts reach the CPU
    - prevents devices from injecting arbitrary interrupts

- Why IIRC matters:
  - ensures interrupts are delivered to the correct VM / process
  - prevents denial-of-service or privilege escalation via interrupts

- **Key idea:**
  - MMU $\rightarrow$ controls CPU memory accesses
  - IOMMU $\rightarrow$ controls device DMA
  - IIRC $\rightarrow$ controls device interrupts

VM and Hardware: Conclusion

Pros:

- Allows managing privileged access of virtualized systems
- With little or no overhead
- Allows creating hypervisors capable of applying hardening and integrity checks on the underlying system.

Cons:

- Can be exploited by certain forms of rootkits to make themselves difficult to detect against the virtualized system, annihilating any analysis and remediation effort

Consequently, it is recommended to disable VT-x if you don't need it.

ToC

1. The Great Escape

2. Hardware Assisted Virtualization
2.1 VM Environment
2.2 Hardware Security of System Calls
2.3 Hardware and Hypervisors

3. MicroVM and OCI: The Great Jail
3.1 Overview
3.2 Kata-Containers
3.3 Firecracker

4. Conclusion

ToC

1. The Great Escape

2. Hardware Assisted Virtualization
2.1 VM Environment
2.2 Hardware Security of System Calls
2.3 Hardware and Hypervisors

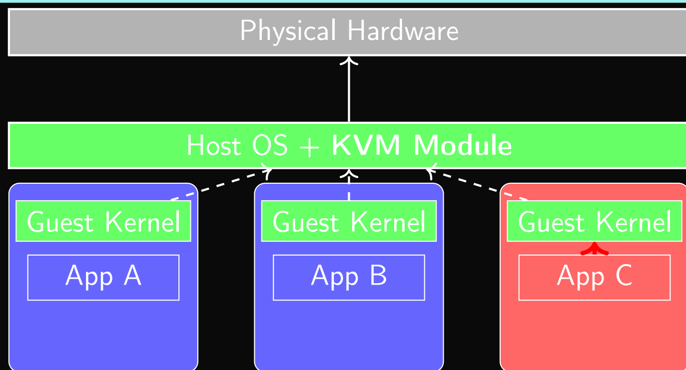
3. MicroVM and OCI: The Great Jail
3.1 Overview
3.2 Kata-Containers
3.3 Firecracker

4. Conclusion

The MicroVM Approach

MicroVM-based Container Architecture

MicroVM-based Containers launch each container (or Kubernetes Pod) inside its own lightweight, optimized **MicroVM** with a dedicated guest kernel.



Key Advantages

- **Strong Security:**
  - Provides the hardware-backed isolation of a full VM.
  - Eliminates the shared kernel attack vector.
- **High Performance:**
  - MicroVMs are heavily optimized and boot in milliseconds.
  - Far lighter than traditional, general-purpose VMs.
- **Full Compatibility:**
  - Fully OCI-compliant runtime.
  - A drop-in replacement for ‘runc‘ in Kubernetes (via CRI) and Docker.
- **Flexible Workloads:**
  - Allows you to safely run untrusted or "noisy neighbor" workloads on the same physical host as your critical applications.

ToC

1. The Great Escape

2. Hardware Assisted Virtualization
2.1 VM Environment
2.2 Hardware Security of System Calls
2.3 Hardware and Hypervisors

3. MicroVM and OCI: The Great Jail
3.1 Overview
3.2 Kata-Containers
3.3 Firecracker

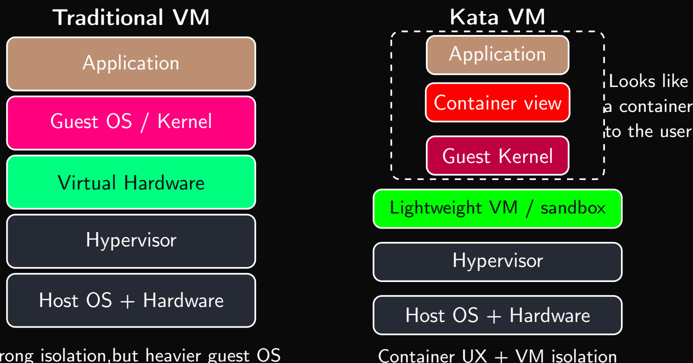
4. Conclusion

How Kata-Container Works: The Architecture

Kata Containers is an **OCI-compatible runtime**!

- **OCI Runtime Shim (kata-runtime):**
  - Receives commands from Kubernetes/Docker (e.g., "start container").
  - Instead of using ‘runc‘, it creates a new MicroVM.
- **Virtual Machine Monitor (VMM):**
  - Uses a lightweight **QEMU** or **Cloud Hypervisor** to manage the VM and emulate virtual devices (network, disk).
- **Hypervisor (KVM):**
  - The Linux **KVM** module provides the high-performance, hardware-assisted virtualization (HVA) to run the VM’s CPU and memory.
- **Guest Kernel:**
  - A highly optimized, stripped-down Linux kernel that boots in milliseconds inside the MicroVM.

Traditional VM vs Kata VM



- A **Kata VM** uses virtualization, but exposes a **container interface** to the user.
- Kata combines **container ergonomics** with **VM-level isolation**.

Kata Containers

Container + VM : **kata-containers**

- For each container creation, **namespaces** are created to isolate the container from the rest of the processes.
- The kernel of the Kata-VMs are **lightweight**, with a heavily simplified init process.
- Hypervisor is **unchanged**.
- Orchestration logic is **unchanged**.
- Combines the security of **VMs** and the security of **containers**.
- Kata-container == **runtime**!

Kata Containers are micro VMs.

ToC

1. The Great Escape

2. Hardware Assisted Virtualization
2.1 VM Environment
2.2 Hardware Security of System Calls
2.3 Hardware and Hypervisors

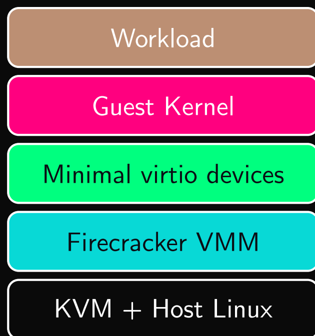
3. MicroVM and OCI: The Great Jail
3.1 Overview
3.2 Kata-Containers
3.3 Firecracker

4. Conclusion

Firecracker

- Open-source microVM runtime
- Developed by Amazon Web Services (AWS)
- Use Cases for Firecracker: serverless computing, scenarios requiring fast and secure isolation.
- Firecracker main design goals:
  - speed, efficiency
  - minimalistic approach to virtualization
  - security

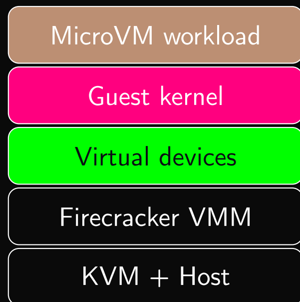
Firecracker



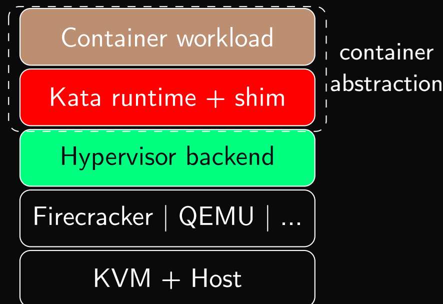
Minimalist VMM for strong isolation with small overhead

- microVMs
- fast startup
- low overhead
- reduced device model

Firecracker vs Kata Containers



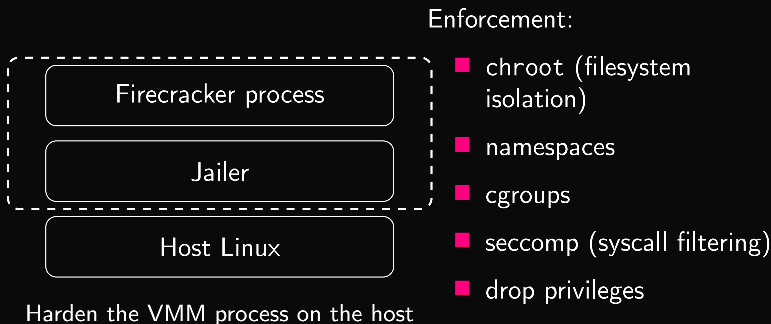
VMM / hypervisor component



container runtime

- Firecracker = **runs microVMs**
- Kata = **runs containers using lightweight VMs**

Firecracker Jailer

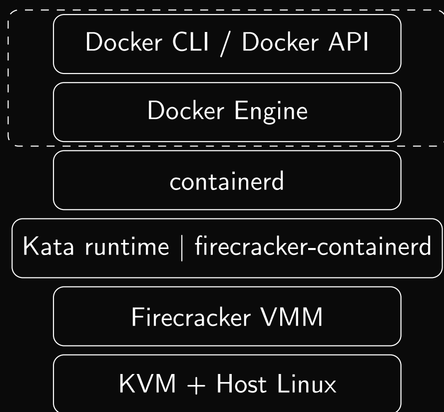


Enforcement:

- chroot (filesystem isolation)
- namespaces
- cgroups
- seccomp (syscall filtering)
- drop privileges

- Jailer isolates **Firecracker itself** with mechanisms similar to traditional container (runc)
- Complements VM isolation (guest side)

Using Firecracker as a Container Runtime

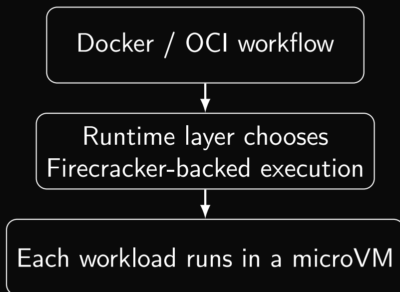


Docker on top, Firecracker underneath

- same Docker workflow
- stronger isolation underneath
- Firecracker is *not* the Docker runtime API itself

- **Direct path:** H-L containerd + L-L `firecracker-containerd`
- **Docker-friendly path:** Docker + Kata, with Kata using Firecracker

Firecracker Integration: Main Idea



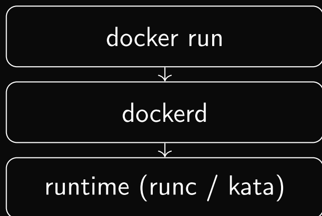
- container UX preserved
- VM isolation added
- higher overhead than runc

Main Idea

Firecracker is usually integrated **below Docker**, as a low-level runtime alternative to runc

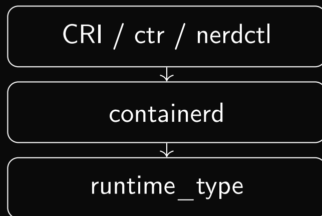
Configuring Runtimes: Docker vs containerd

Docker



- daemon.json
- –runtime=kata

containerd



- config.toml
- runtime_type = io.containerd.kata.v2

Key idea

Same concept, different layer:
Docker configures runtimes at the **daemon level**,
containerd at the **runtime (CRI) level**.

ToC

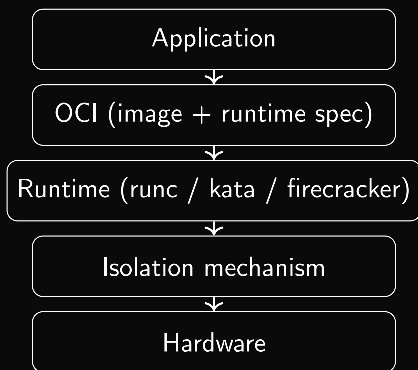
1. The Great Escape

2. Hardware Assisted Virtualization
2.1 VM Environment
2.2 Hardware Security of System Calls
2.3 Hardware and Hypervisors

3. MicroVM and OCI: The Great Jail
3.1 Overview
3.2 Kata-Containers
3.3 Firecracker

4. Conclusion

Recap: Container Execution and Isolation



Same OCI interface, different isolation strategies

- **runc** → namespaces + cgroups
- **Kata** → lightweight VM
- **Firecracker** → microVM

- Containers are **not a single mechanism**
- OCI enables **pluggable execution models**
- The one we discussed are not the only ones...

gVisor: A Different Isolation Approach

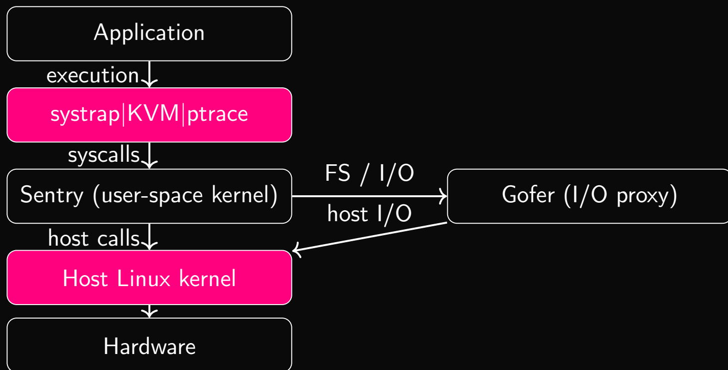
- Developed by **Google** (2018)
- Sandboxed container runtime
- Implements a **user-space kernel**
- Intercepts system calls
- reduces kernel attack surface
- no VM, no hardware virtualization

- No direct access to host kernel
- Alternative to both **runc** and microVMs

Comparison with Other Solutions

- Containers → host kernel
- gVisor → user-space kernel
- MicroVM → virtualized kernel

gVisor Architecture



- Platform = systrap|KVM|ptrace = **how execution is trapped**
- Sentry = **what emulates the kernel**

What If the Host Cannot Be Trusted?

- So far, we assume:
  - the host kernel is trusted
  - the hypervisor is trusted
- But in multi-tenant environments:
  - cloud providers run the infrastructure
  - users may not trust the host
- New approach: **Confidential Computing**
  - isolate workloads even from the host
  - protect memory with hardware encryption
- Examples:
  - Intel TDX
  - AMD SEV

Quiz Time!