Building a Firmware for Virtual Machines using Rust

Robert Bradford
Intel

Project:
Rust Hypervisor Firmware

Motivation

Why Rust?

- New language with a focus on correctness and performance
- Compiled to native code offering performance similar to C
- Memory management without garbage collection
- Designed for systems programming

Why A New Hypervisor Firmware?

A new hypervisor deserves a new firmware!

Cloud Hypervisor objectives:

- High performance type-2 VMM using KVM
- Minimal hardware emulation → small attack surface
- Suitable for use with Kata Containers
- Suitable as a “pet” virtual machine monitor (with persistent storage, networking and a generic operating system)

Why Not OVMF?

- OVMF is a TianoCore based UEFI firmware – used with NEMU and QEMU
- Experience of OVMF from porting NEMU “virt” machine type:
- “Legacy” hardware expectations
- Full featured → complex
- Linux cloud workloads main focus
- Want compatibility with Linux loader

Linux Loader Compatibility

- All Rust based hypervisors have an ELF loader for the Linux kernel
- Used to load uncompressed vmlinux kernel
- Boots in long mode with identity mapping
- LDT & GDT setup
- Provides an E820 table with the memory layout

Why A Firmware Then?

- If can already load a Linux kernel with the hypervisor…why this project?
- Direct loading is perfect for: container style workloads (e.g. Kata Containers or Firecracker) or full control of the stack (e.g. Crostini on Chrome OS)
- For wider cloud use cases: End-user need to control their own boot (e.g. to update kernel)

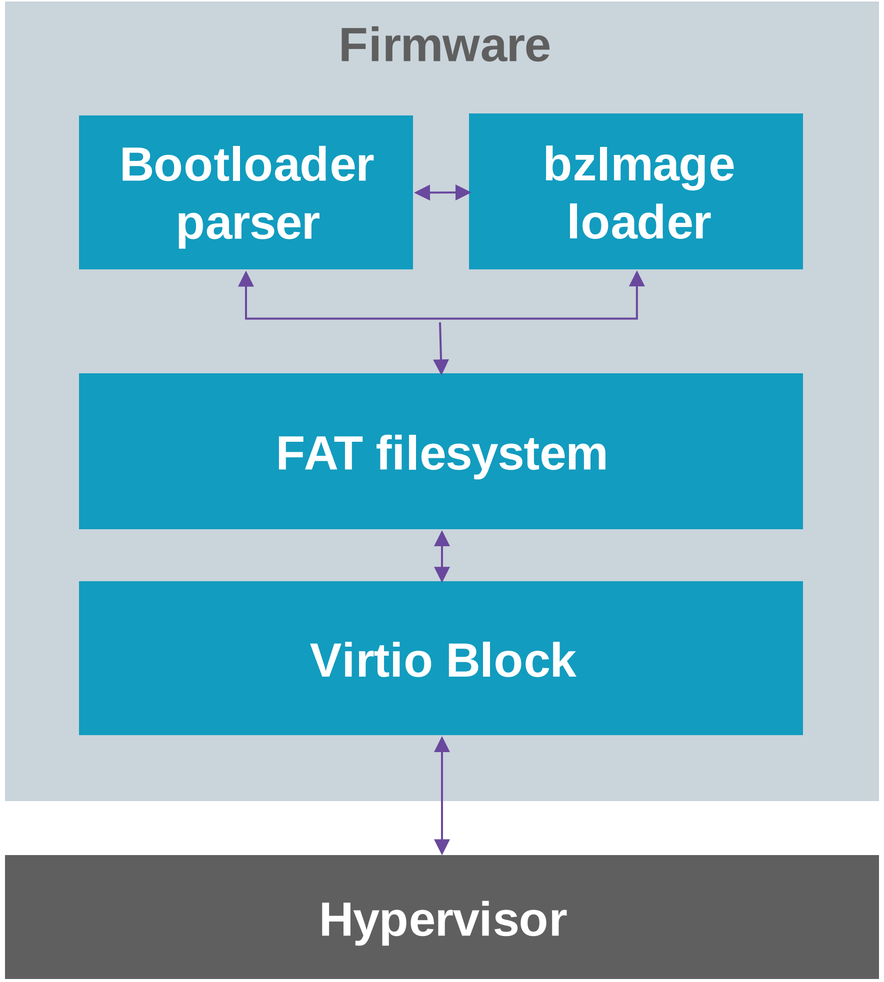
Architecture

Architecture

- Two modes of operation:
- FreeDesktop loader
  - Used for ClearLinux
- EFI loading
  - Used for Debian and Ubuntu

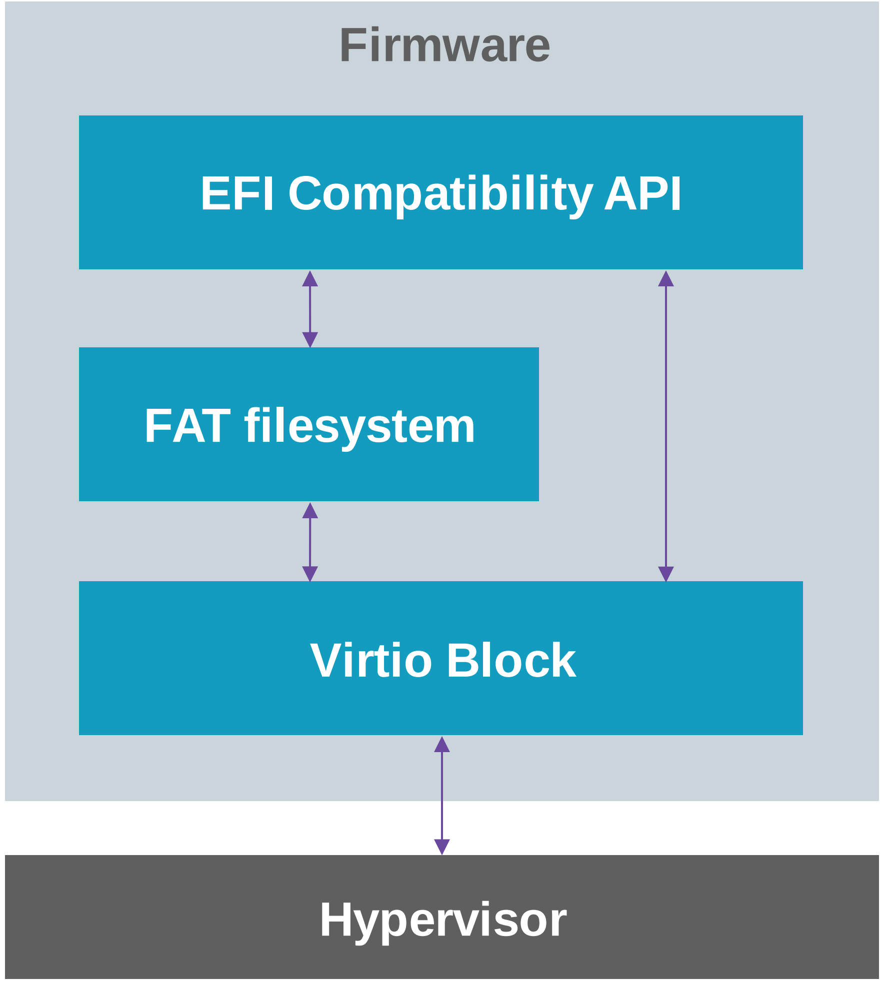
FreeDesktop Loading

- Virtio transport: MMIO and PCI
- Block device driver (virtio-blk)
- GPT partition parsing
- FAT filesystem implementation
- bzImage loader
- FreeDesktop bootloader specification parser



EFI Loading

- Virtio transport: MMIO and PCI
- Block device driver (virtio-blk)
- GPT partition parsing
- FAT filesystem implementation
- PE32+ loader
- "EFI Compatibility" API



Basic setup

- Loaded by hypervisor at 0x100000 (1MiB)
- Establishes wider identity mapping
- Parses kernel command line for MMIO block device details
- Reads E820 table from zero page
- Probes block device and searches for filesystem

FreeDesktop loader

- Parses FreeDesktop bootloader specification configuration
- Loads bzImage via 64-bit bootloader protocol at 0x200000 (2 MiB)
- Loads initrd and populates command line
- Updates zero page with new details per spec (including revised E820)
- Jumps into kernel at 64 bit entry
- No more interaction with firmware

EFI image loader

- PE32+ loader
- “EFI compatibility” layer
- Uses “r-efi” crate – definition of common EFI structures in Rust
- EFI memory allocator
- Filesystem + block abstraction
- Able to boot Linux kernel built with CONFIG_EFI_STUB
- Boots shim + GRUB as used by Ubuntu image
- Not aiming for full EFI functionality

Evaluation

Evaluation of Rust (for Firmware)

Memory safety - *Helps* avoid many classes of security issues

But … firmware needs fine grained control of memory

Ergonomic - great editor support, unit testing in the box, powerful build system

But … custom target for linker script, need to use “core”, “nightly” compiler

Flexible - have control over *some* low-level details

But … firmware patterns pushes Rust language to its limits

High performance - almost native performance

Community - wide community developing firmware, operating systems and other low-level components in Rust

Conclusion

Development Status

- Experiment. Not for production!
- Currently developed and tested against Firecracker and Cloud Hypervisor
- Apache 2.0 licensed
- On GitHub: https://github.com/intel/rust-hypervisor-firmware
- External contributions welcome!

Q&A

Disclaimers

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request. No product or component can be absolutely secure.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting www.intel.com/design/literature.htm

Intel, the Intel logo are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.

Other names and brands may be claimed as the property of others

© Intel Corporation.