

Kage: device driver isolation in the Linux kernel with LFI

Zachary Yedidia (Stanford & Google), Nic Watson (Google), Tal Garfinkel (Google)



Device Drivers: A Frequent Source of Exploits

Android top targets:

- WiFi (accessible from network)
- GPU (accessible from browser)

Driver LoC (Linux):

- bcmdhd: ~300k (WiFi)
- iwlwifi: ~240k (WiFi)
- Mali kbase: ~100k (GPU)
- All WiFi: 1.5M+
- All GPU: 2M+

MediaTek components

These vulnerabilities affect MediaTek components and further details are available directly from MediaTek. The severity assessment of these issues is provided directly by MediaTek.

CVE	References	Severity	Subcomponent
CVE-2024-20154	A-376809176 M-MOLY00720348 ↗	Critical	Modem
CVE-2024-20146	A-376814209 M-ALPS09137491 ↗	High	wlan
CVE-2024-20148	A-376814212 M-ALPS09136494 ↗	High	wlan
CVE-2024-20105	A-376821905 M-ALPS09062027 ↗	High	m4u
CVE-2024-20140	A-376816308 M-ALPS09270402 ↗	High	power
CVE-2024-20143	A-376814208 M-ALPS09167056 ↗	High	DA
CVE-2024-20144	A-376816309 M-ALPS09167056 ↗	High	DA

MediaTek components

These vulnerabilities affect MediaTek components and further details are available directly from MediaTek. The severity assessment of these issues is provided directly by MediaTek.

CVE	References	Severity	Subcomponent
CVE-2025-20634	A-381773169 M-MOLY01289384 ↗	High	Modem
CVE-2024-20141	A-381773173 M-ALPS09291402 ↗	High	DA
CVE-2024-20142	A-381773175 M-ALPS09291406 ↗	High	DA
CVE-2025-20635	A-381771695 M-ALPS09403752 ↗	High	DA
CVE-2025-20636	A-381773171 M-ALPS09403554 ↗	High	secmem

and are described in further detail in the appropriate Qualcomm assessment of these issues is provided directly by Qualcomm.

Severity	Subcomponent
Critical	WLAN
High	WLAN
High	Camera
High	Camera
High	Camera
High	Camera

components and are described in further detail in the appropriate Qualcomm severity assessment of these issues is provided directly by Qualcomm.

References	Severity	Subcomponent
-350500921 IC-CR#3445828	High	Kernel
-372003017 IC-CR#3845043 [2]	High	Kernel
-372002616 IC-CR#3852338	High	WLAN

Device Drivers: A Frequent Source of Exploits

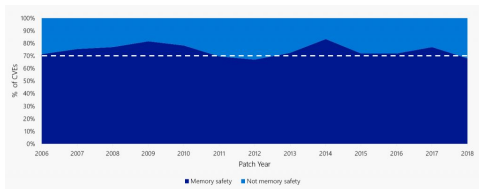
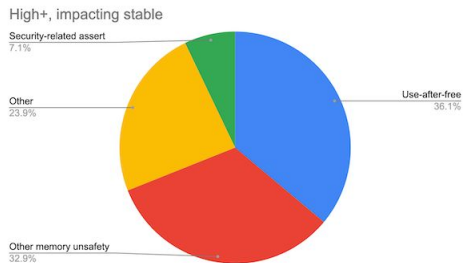
More broadly: how can we build more secure systems from existing monolithic architectures?

Increased urgency now in the age of AI

Sandboxing: a Tool for Memory Safety

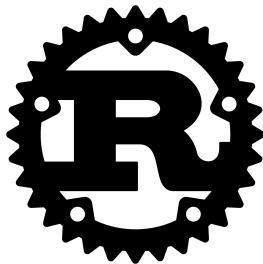
Critical Vulnerabilities

Memory-safety bugs represent the vast majority of high-severity and critical security vulnerabilities.



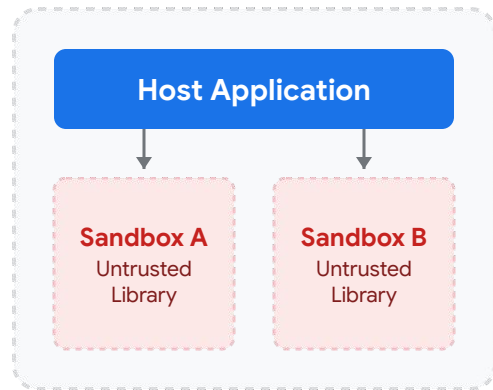
Safe Language Rewrite

Rewriting code in a memory-safe language is the ideal solution, but is not always feasible or available.



Sandboxing

An effective mechanism to apply safety without rewriting. Well-suited for examples such as media codecs, parsers, and compression.



Talk Outline

Part 1

LFI: a newly-available compiler-based mechanism for sandboxing

Part 2

Isolating simple kernel modules with LFI

Part 3

Picking a better isolation boundary for large device drivers

LFI: A Sandboxing Toolchain in LLVM



Performant & Compatible

Sandboxes native and legacy code in-process with minimal overhead and broad software compatibility.

Simple & Flexible

A lightweight, unified mechanism designed to isolate libraries without introducing significant new architectural complexity.

Built into LLVM

Coming soon (or already available) to an LLVM near you!

Used in Android

For isolating audio codecs in the media subsystem.

LFI: Feature and Performance Overview

Architecture Support: [AArch64](#) and [x86-64](#)

LFI 4GiB SPEC 2017 Overhead

Full (R/W) Isolation:

AArch64: 6.9% – 8.6% | x86-64: 7.1% – 7.9%

Stores-Only Isolation:

AArch64: 1.1% – 1.9% | x86-64: 3.8% – 5.3%

Jumps-Only Isolation:

AArch64: 0.5% – 0.8% | x86-64: 3.2% – 4.3%

Lightweight Transition Cost

Fast Transitions:

- Tens of nanoseconds.
- Orders of magnitude faster than a process context switch on Linux.

In-Process Isolation:

- Does not require a separate process.
- Address space is shared with the host.

How it works: Rewriting and Verification

Architecture & Mechanism

New architecture target: **aarch64_lfi**

(example: aarch64_lfi-linux-musl)

Uses a hook in **MCStreamer** to apply rewrites

Goal: Automatically handle hand-written assembly

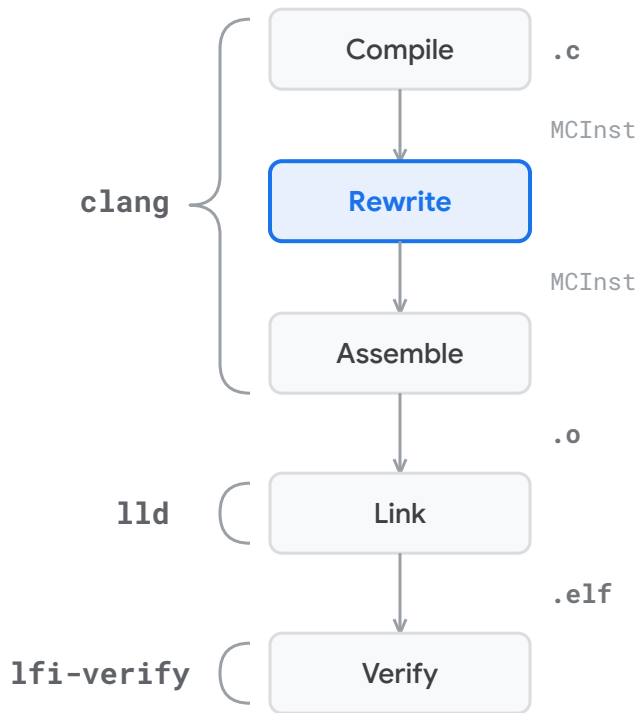
Previously: External .s/.o rewriter

Other compiler stages remain unchanged*

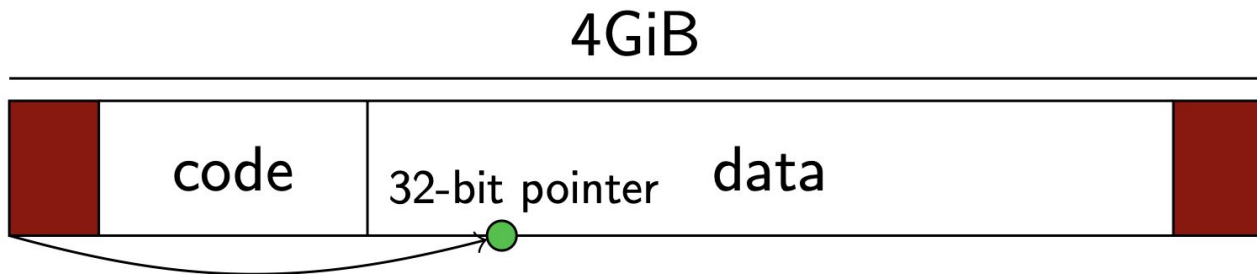
*Only need to reserve a few registers

All sandboxed code goes through the rewriter:

Libc, dynamic linker, libc++, compiler-rt, ...



How it works: Runtime Model



Runtime handles syscalls (indirect branches) and enforces W^X.

Dynamic codegen can be supported: Run verifier before `mprotect(PROT_EXEC)`.

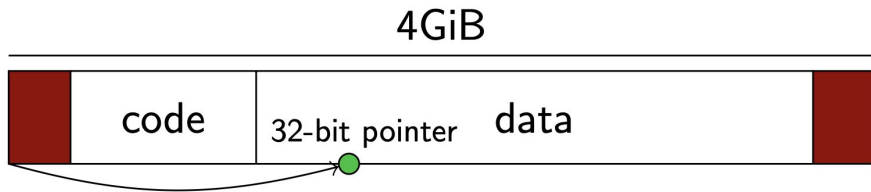
48-bit address space: supports up to 64K sandboxes (80KiB guard pages).

64-bit ABI: matches host ABI, compatible with existing code.

AArch64 Overview

Reserve x27: sandbox base.

Reserve x28: any sandbox address.



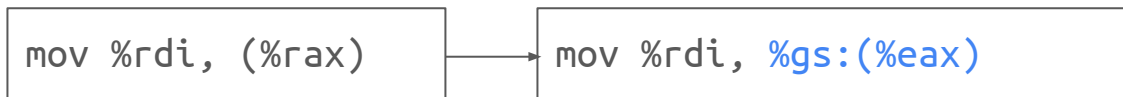
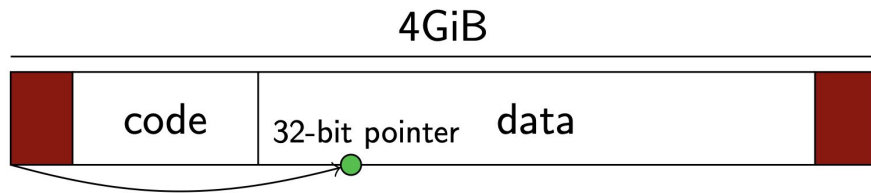
Mask with addressing mode

Covered in more detail in last year's talk.

X86-64 Overview

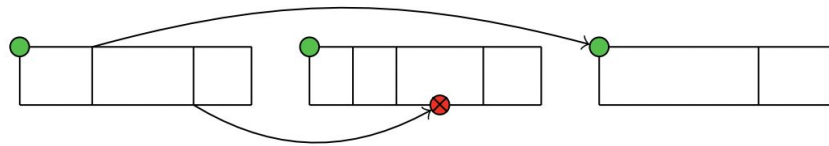
Reserve %gs: sandbox base.

- Critical for performance.



Mask with addressing mode

Control flow: requires *bundling* to prevent execution of instructions inside other instructions (like Native Client, PittSFeld).



Optimization: apply bundle padding using ignored instruction prefixes.

LLVM Upstream Status

AArch64 Support

Fully upstream

- ✓ MCLFIRewriter infrastructure
- ✓ aarch64_lfi target triple
- ✓ System rewrites
- ✓ Control-flow rewrites
- ✓ Memory rewrites
- ✓ PAC support

X86 Support

Ongoing (ETA 1 month, LLVM 24 release cycle)

- ✓ x86_64_lfi target triple
- ✓ System rewrites
- ✓ Bundling
- ✓ Control-flow rewrites
- ... Memory rewrites

Thank you to LLVM reviewers: Nick Desaulniers, Peter Smith, Fangrui Song, Alexis Engelke, Eli Friedman, Vitaly Buka, Phoebe Wang

Idea: Use LFI in the Kernel

We have this new isolation capability, would it be useful to kernel-space in addition to user-space?

Compiler-based

LFI avoids conflicts with address-space-based isolation infrastructure

Fast transitions

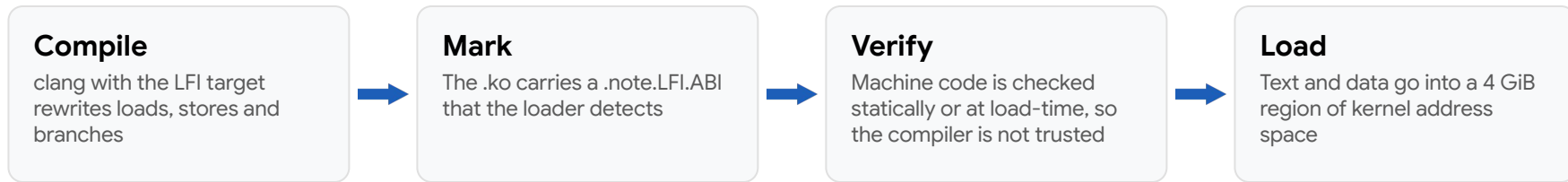
Useful when retrofitting isolation on a boundary that previously assumed function-call-level transition performance

Fine-grained

Isolation can be fine-grained, and even potentially allow certain privileged instructions

Initial direction: use LFI to build infrastructure for isolated kernel modules

Building Kernel Code with LFI



```
make M=$PWD CC=clang EXTRA_CFLAGS="--target=aarch64_lfi-linux-gnu"
```

Same kbuild

An out-of-tree module build with a different compiler and one target flag. Native and LFI variants build from the same source.

No privileged instructions

Since module runs at EL1, the verifier rejects MSR/MRS, TLBI, DAIF changes and ERET.

Sandbox set up by loader

The runtime components of the sandbox, such as the virtual address space, are automatically set up and configured by the module linker/loader.

Kage: LFI Sandboxes for Kernel Modules

Kernel (host)

Module loader: detects the LFI note, creates the sandbox

Kernel API: `kmalloc`, `device_*`, `printk`, ...

Object storage: kernel pointers become opaque handles

Guest to host



Trampoline, then a guard generated from the BTF signature

Host to guest



`kage_call`: function pointer arguments wrapped as closures

LFI sandbox, 4 GiB

`module.ko`, built with the LFI target

Guest heap, stacks and scratch memory

Holds handles instead of kernel pointers

Where Kage Works

michael_mic

Hashing algorithm

The kernel calls shash ops in the guest. Keys, data and digests are copied in and out.

rtc-test

Simple device driver

Device registration through guarded devm and device_* calls, with handles for kernel objects.

chardev

Character device

A tasklet callback enters the guest from softirq context.

A good fit when the interface is

✓ Narrow: a few typed calls

✓ Copyable: data in, data out

✓ Free of shared mutable state

Candidates: parsers and decoders, compression algorithms, small platform drivers.

Scaling Up: NVMe, Network, WiFi, GPU

Problem: Larger drivers became intractable to sandbox with Kage.

Large Symbol Surface

Imports many kernel symbols:

- `nvme.ko`: 181 symbols imported
- `iwlmvm.ko`: 354 symbols imported

Shared Mutable State

Coupled data structures across boundary:

- Raw kernel pointers
- Shared locks
- Reference counts

Unstable Kernel API

Explicit policy against stable in-kernel ABIs.

Internal kernel interfaces shift between versions.

Trust Assumptions

Code written assuming full kernel trust.

Lacks defensive checks required for sandboxed execution.

Scaling Up: NVMe, Network, WiFi, GPU

Problem: Larger drivers became intractable to sandbox with Kage.

Large Symbol Surface

Imports many kernel symbols:

- `nvme.ko`: 181 symbols imported
- `iwlmvm.ko`: 354 symbols imported

Shared Mutable State

Coupled data structures across boundary:

- Raw kernel pointers
- Shared locks
- Reference counts

Unstable Kernel API

Explicit policy against stable in-kernel ABIs.

Internal kernel interfaces shift between versions.

Trust Assumptions

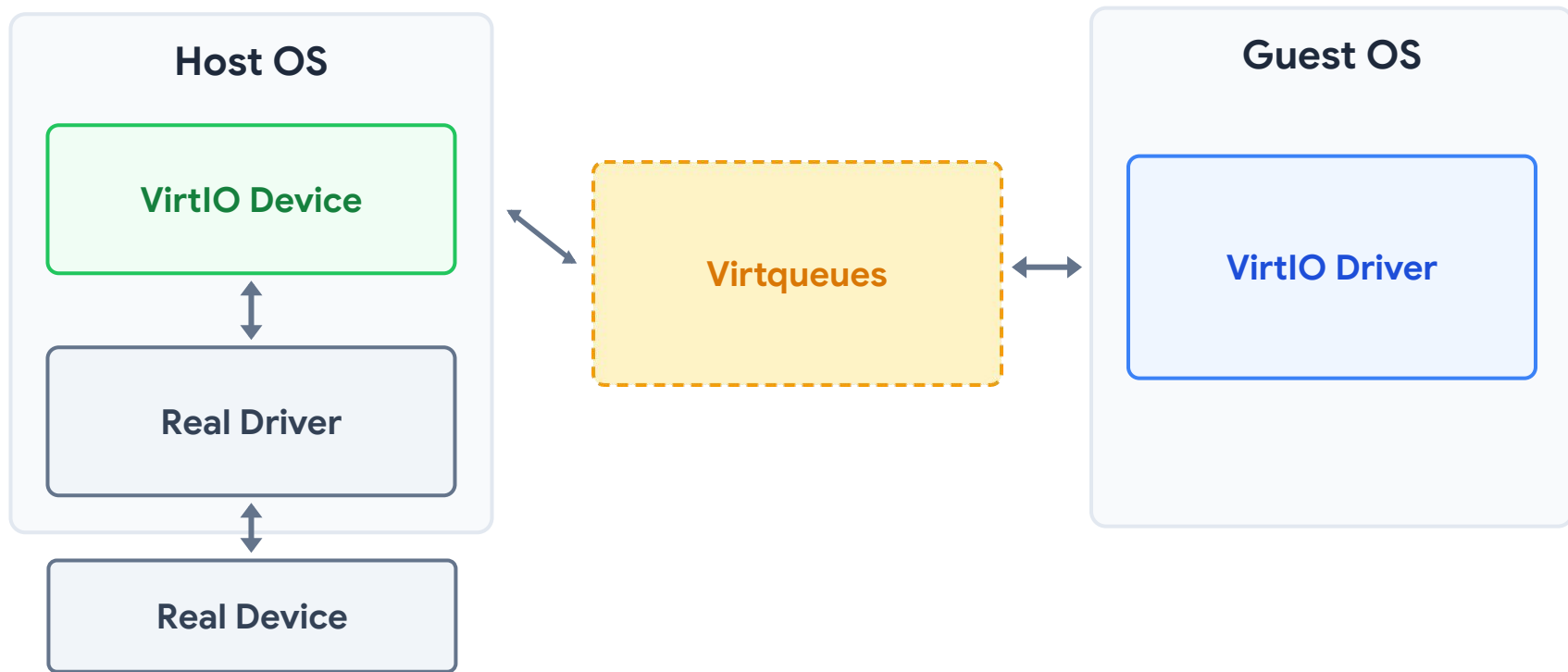
Code written assuming full kernel trust.

Lacks defensive checks required for sandboxed execution.

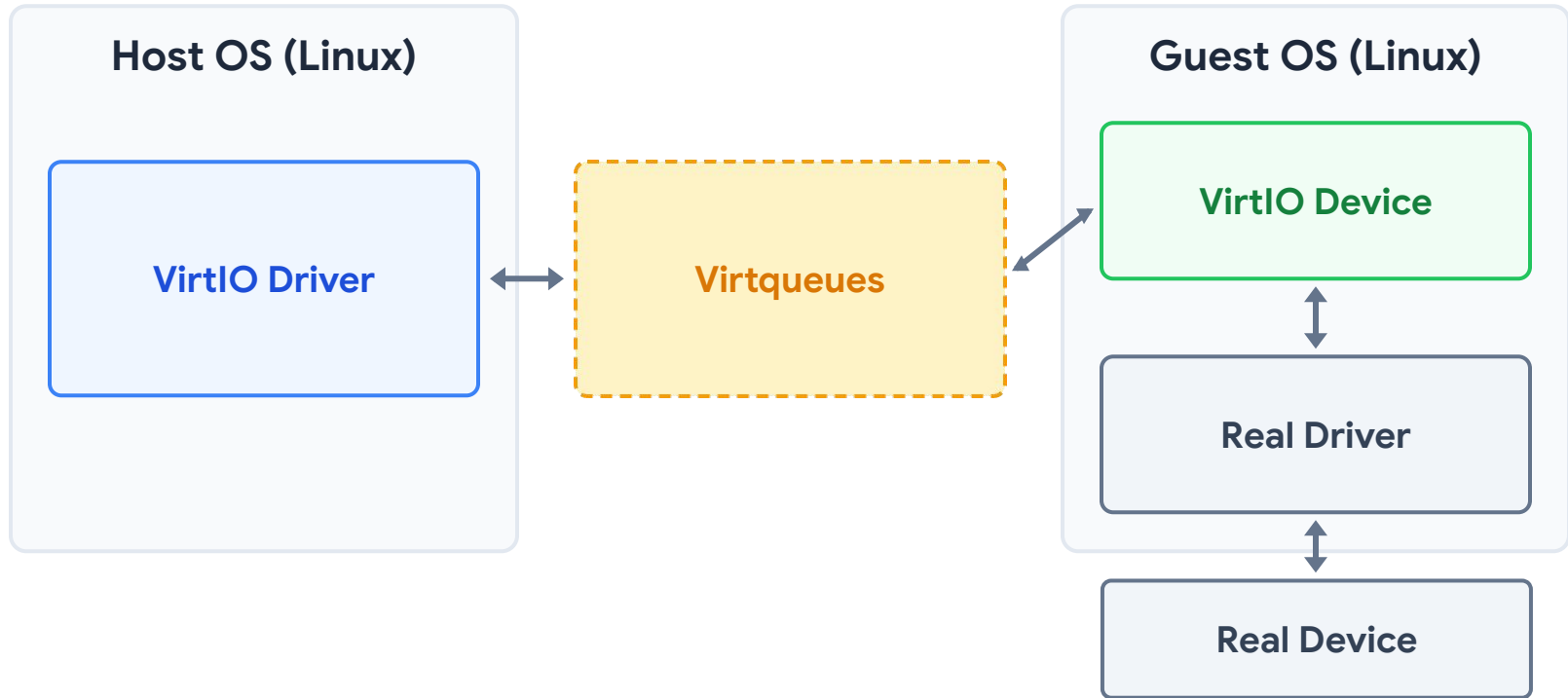
Idea: Investigate the APIs that exist between drivers and (emulated) devices.

VirtIO: a Stable Driver API?

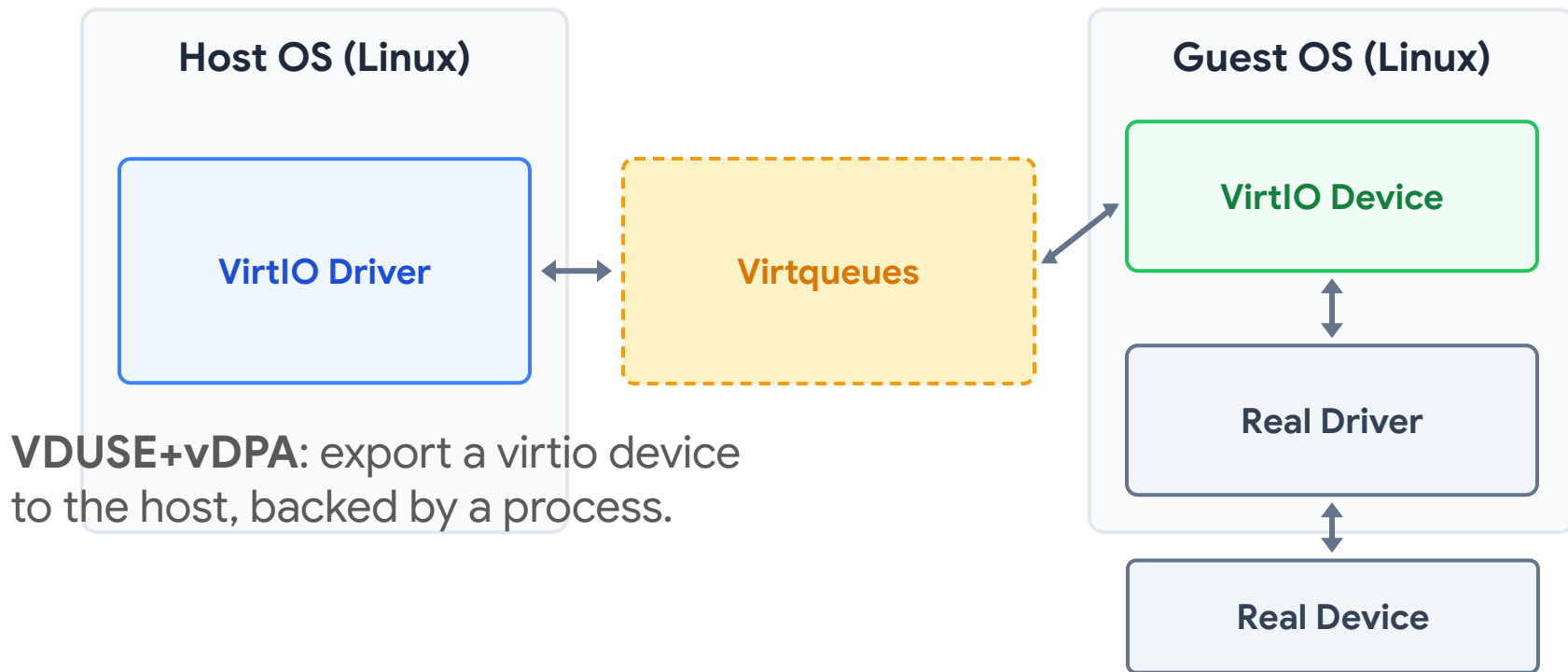
Typical use-case: expose performant and uniform devices to a guest VM.



VKage Architecture: Inverted VirtIO

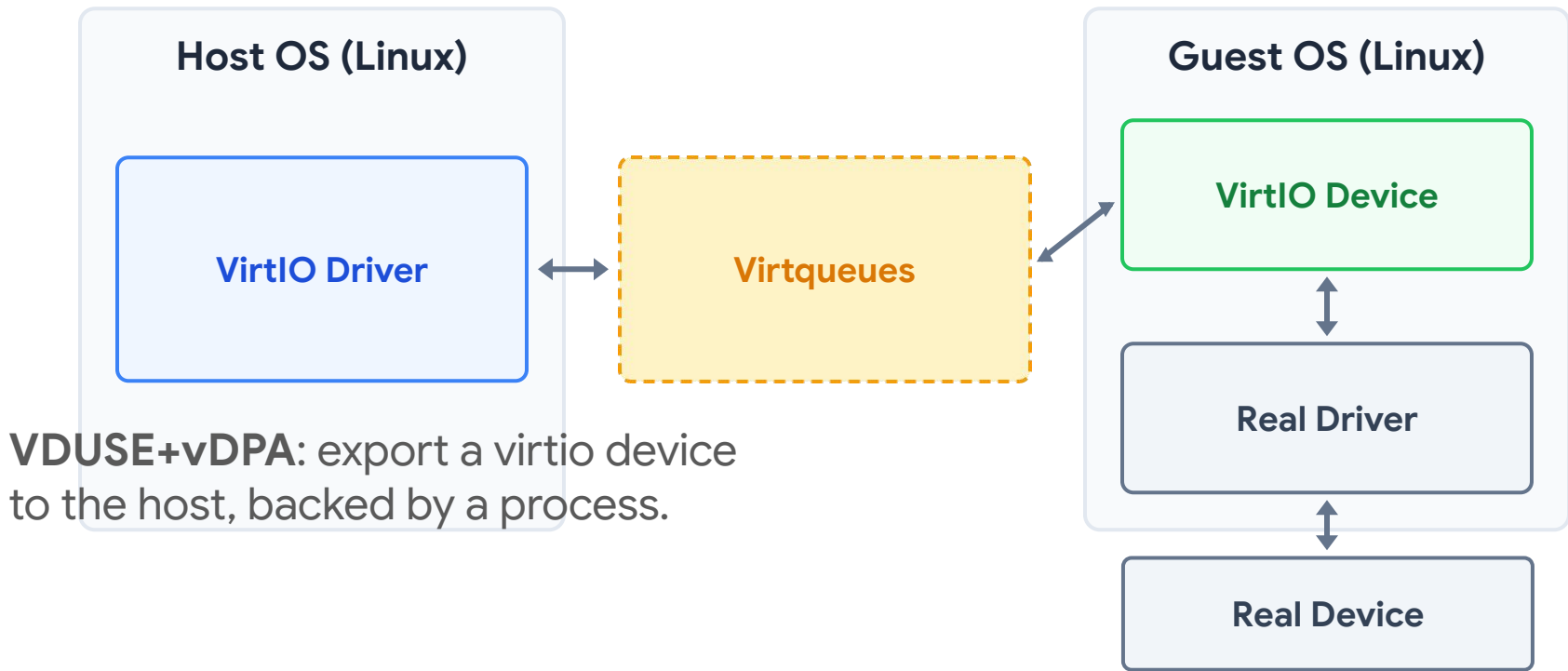


VKage Architecture: Inverted VirtIO



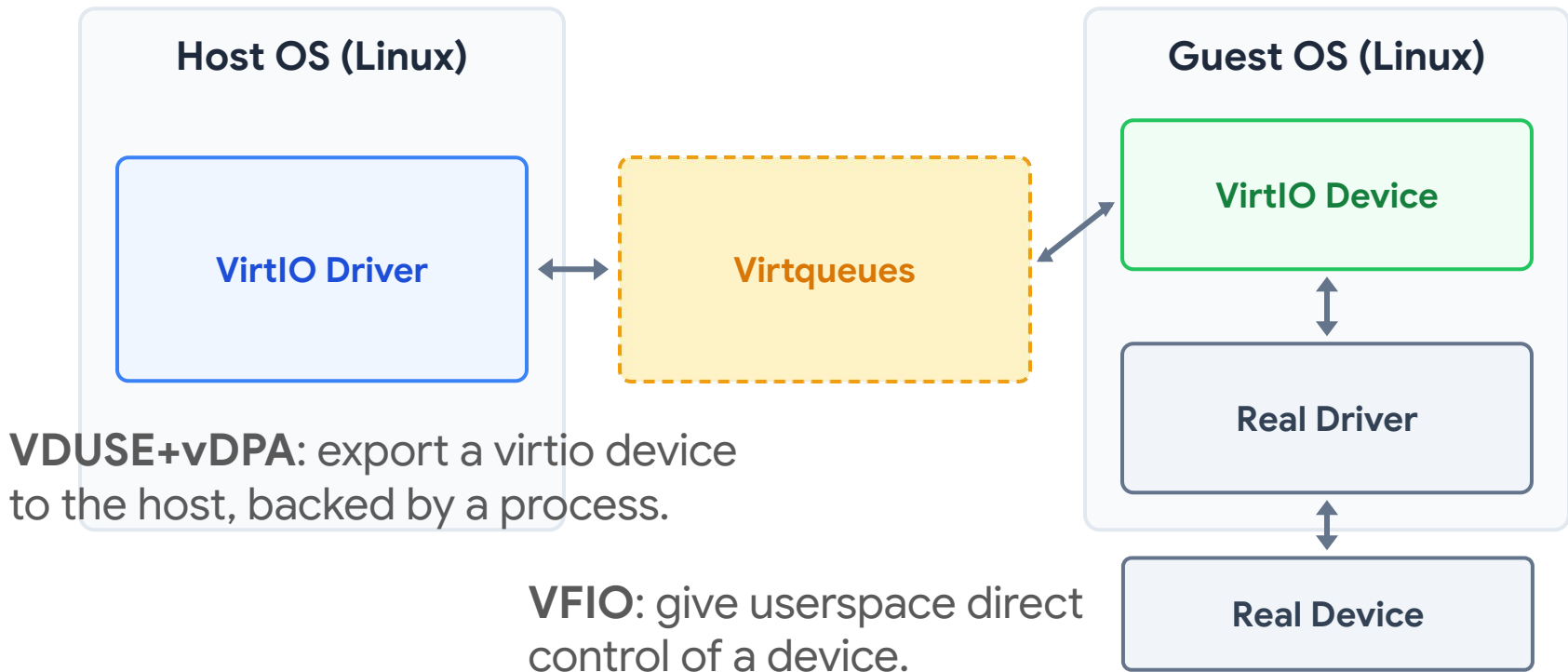
VKage Architecture: Inverted VirtIO

Usermode Linux (UML): run Linux in a userspace process.



VKage Architecture: Inverted VirtIO

Usermode Linux (UML): run Linux in a userspace process.



VKage Prototypes

Working Prototypes

NVMe

Disk devices via virtio-blk

Network (igb)

Ethernet devices via virtio-net

WiFi (iwlwifi & mt7925)

Wireless devices via virtio-wlan (custom)

Next Steps

Performance Measurements

Benchmarking and optimization for target use-cases

Android Device Drivers

Test and validate implementation on Android device

Can this approach work for GPUs?

In principle, yes, but needs more work to determine for real, and measure performance impact

Limits of VirtIO



Missing Device Classes

We had to write a custom `virtio_wlan` driver.



Device-Specific Features

VirtIO exposes a generic device interface.



High Latency

Every request crosses into a user process and back.

Future: LFI+VirtIO?

VirtIO Confinement Boundary

1. Give a device to userspace using **VFIO**.
2. Control the VFIO device with a **User-Mode Linux (UML) process** running the real driver.
3. Expose a **virtio device** on host and use **VDUSE** to back it with the UML process.

LFI Kernel Module Loader

- Prototype can load and execute **LFI'd kernel modules** with mediated host callbacks.
- Uses **BTF (BPF Type Format)** for binding functions dynamically.
- Useful for sandboxing small components.

Combined Approach: Run a UML driver in an LFI'd kernel module

May be useful for latency-sensitive drivers.

Thank you!

Questions?