Mocking Linux Driver Code for VR

- Why do drivers *still* have bugs?

Bernie Lampe, Ph.D.

Sept 6, 2024

Who am I?

- Bernie Lampe
  - Vice President of Research, Graykey Labs
  - Offensive Security Researcher (mostly Linux and Android)
  - CV: https://bernardlampe.com
- Graykey Labs by Magnet Forensics
  - Provide access and extract digital evidence for law enforcement

MAGNET FORENSICS

Why am I here?

- How does one find the bugs?
  - Lots written about exploitation methods, write-ups usually start with bug
- Ideal steps to finding the bugs:
  - Collect, select attack surfaces, audit, reverse, build, debug, write parsers for inputs, emulate, instrument, write tests, fuzz test, reproduce n-days
  - Active process but not exact science
- How would one perform offensive research on kernel drivers?
  - Pick good attack surfaces
  - Emulation and instrumentation steps are harder

Why are co-processer drivers a good target?

- Good attack surface qualities:
  - Complex code base
  - Resource sharing
  - Accessible via MAC and DAC constraints
  - Consistency across Android vendors
  - Slowly and steadily changing
  - Discrete release revisions
  - Not written by the core Linux developers
  - Proprietary fewer eyeballs

- Few surfaces fit all these qualities better than the co-processors

What is difficult about drivers?

- Emulation and instrumentation are harder
  - Not necessarily open source or revisioned code
  - No introspection from the kernel or hardware side on production devices

- What are the options?
  - Black box fuzz using production devices with little instrumentation
  - Stare at the artifacts you do have - audit and reverse the driver / firmware
  - Create virtual devices in an emulator such as Qemu
  - “Find” development devices and supporting software / hardware / docs
    - Thundercomm boards as example

Is there a middle ground with less work?

- Driver development usually starts “pre-silicon” to not block the development of supporting userspace libraries
- Model hardware behavior using GEM5 emulator

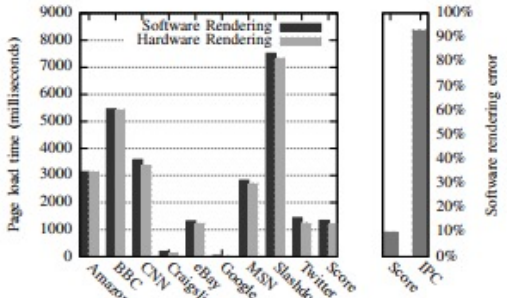
NoMali: Simulating a Realistic Graphics Driver Stack Using a Stub GPU

René de Jong
ARM Research
Cambridge
rene.dejong@arm.com

Andreas Sandberg
ARM Research
Cambridge
andreas.sandberg@arm.com

***Abstract*—Since the advent of the smartphone, all high-end mobile devices have required graphics acceleration in the form of a GPU. Today, even low-power devices such as smart-watches use GPUs for rendering and composition. However, the computer architecture community has largely ignored these developments when evaluating new architecture proposals.**

A common approach when evaluating CPU designs for the mobile space has been to use software rendering instead of a GPU model. However, due to the ubiquity of GPUs in mobile devices, they are used in both 3D applications *and* 2D applications. For example, when running a 2D application such as the web browser in Android with a software renderer instead of a GPU, the CPU ends up executing twice as many instructions. Both the CPU characteristics and the memory



https://github.com/ARM-software/nomali-model
https://ieeexplore.ieee.org/document/7482100

Where to start in 2020 versions r0p0 - r29p0?

- Download the Mali driver
  - Appreciation for the discrete release numbers and open source – thx ARM
- Check out the driver “user manual” (i.e. Kconfig)
  - driver/product/kernel/drivers/gpu/arm/midgard/Kconfig

```
config MALI_NO_MALI
    bool "No Mali"
    depends on MALI_MIDGARD && MALI_EXPERT
    default n
    help
      This can be used to test the driver in a simulated environment
      whereby the hardware is not physically present. If the hardware is physically
      present it will not be used. This can be used to test the majority of the
      driver without needing actual hardware or for software benchmarking.
      All calls to the simulated hardware will complete immediately as if the hardware
      completed the task.
```

Where to start in 2020 versions r0p0 - r29p0?

```
user@host:/tmp/t$ wget https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.19.320.tar.xz >/dev/null 2>&1
user@host:/tmp/t$ wget https://developer.arm.com/-/media/Files/downloads/mali-drivers/kernel/mali-bifrost-gpu/BX304L01B-SW-99002-r29p0-01eac0.tar > /dev/null 2>&1
user@host:/tmp/t$ tar xf linux-4.19.320.tar.xz
user@host:/tmp/t$ tar xf BX304L01B-SW-99002-r29p0-01eac0.tar
user@host:/tmp/t$ cp -r driver/product/kernel/drivers/gpu/arm ./linux-4.19.320/drivers/gpu/
user@host:/tmp/t$ cp driver/product/kernel/include/linux/* ./linux-4.19.320/include/linux/
user@host:/tmp/t$ sed -i "/^obj-y/ s/$/ arm\//" linux-4.19.320/drivers/gpu/Makefile
user@host:/tmp/t$ sed -i '/.*drm\/Kconfig"/a source "drivers\/gpu\/arm\/Kconfig"' linux-4.19.320/drivers/video/Kconfig
user@host:/tmp/t$ cat <<- EOF >> ./linux-4.19.320/arch/x86/configs/x86_64_defconfig
CONFIG_MALI_MIDGARD=y
CONFIG_MALI_NO_MALI=y
CONFIG_MALI_DDK_VERSION=y
CONFIG_MALI_PLATFORM_NAME="devicetree"
CONFIG_MALI_EXPERT=y
CONFIG_MALI_DEBUG=y
CONFIG_MALI_FENCE_DEBUG=y
CONFIG_MALI_PRFCNT_SET_PRIMARY=y
CONFIG_MALI_GATOR_SUPPORT=n
CONFIG_MALI_MIDGARD_ENABLE_TRACE=n
CONFIG_MALI_SYSTEM_TRACE=n
CONFIG_MALI_KUTF=n
CONFIG_MALI_IRQ_LATENCY=n
CONFIG_CONFIG_MALI_CLK_RATE_TRACE_PORTAL=n
EOF
```

Download the appropriate kernel and driver. Kernel 4.19 and driver version 29 both circa 2020.

Where to start in 2020 versions r0p0 - r29p0?

```
user@host:/tmp/t$ wget https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.19.320.tar.xz >/dev/null 2>&1
user@host:/tmp/t$ wget https://developer.arm.com/-/media/Files/downloads/mali-drivers/kernel/mali-bifrost-gpu/BX304L01B-SW-99002-r29p0-01eac0.tar > /dev/null 2>&1
user@host:/tmp/t$ tar xf linux-4.19.320.tar.xz
user@host:/tmp/t$ tar xf BX304L01B-SW-99002-r29p0-01eac0.tar
user@host:/tmp/t$ cp -r driver/product/kernel/drivers/gpu/arm ./linux-4.19.320/drivers/gpu/
user@host:/tmp/t$ cp driver/product/kernel/include/linux/* ./linux-4.19.320/include/linux/
user@host:/tmp/t$ sed -i "/^obj-y/ s/$/ arm\//" linux-4.19.320/drivers/gpu/Makefile
user@host:/tmp/t$ sed -i '/.*drm\/Kconfig"/a source "drivers\/gpu\/arm\/Kconfig"' linux-4.19.320/drivers/video/Kconfig
user@host:/tmp/t$ cat <<- EOF >> ./linux-4.19.320/arch/x86/configs/x86_64_defconfig
CONFIG_MALI_MIDGARD=y
CONFIG_MALI_NO_MALI=y
CONFIG_MALI_DDK_VERSION=y
CONFIG_MALI_PLATFORM_NAME="devicetree"
CONFIG_MALI_EXPERT=y
CONFIG_MALI_DEBUG=y
CONFIG_MALI_FENCE_DEBUG=y
CONFIG_MALI_PRFCNT_SET_PRIMARY=y
CONFIG_MALI_GATOR_SUPPORT=n
CONFIG_MALI_MIDGARD_ENABLE_TRACE=n
CONFIG_MALI_SYSTEM_TRACE=n
CONFIG_MALI_KUTF=n
CONFIG_MALI_IRQ_LATENCY=n
CONFIG_CONFIG_MALI_CLK_RATE_TRACE_PORTAL=n
EOF
```

Copy the driver code and headers into the kernel source tree. This is an in-tree build.

Where to start in 2020 versions r0p0 - r29p0?

```
user@host:/tmp/t$ wget https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.19.320.tar.xz >/dev/null 2>&1
user@host:/tmp/t$ wget https://developer.arm.com/-/media/Files/downloads/mali-drivers/kernel/mali-bifrost-gpu/BX304L01B-SW-99002-r29p0-01eac0.tar > /dev/null 2>&1
user@host:/tmp/t$ tar xf linux-4.19.320.tar.xz
user@host:/tmp/t$ tar xf BX304L01B-SW-99002-r29p0-01eac0.tar
user@host:/tmp/t$ cp -r driver/product/kernel/drivers/gpu/arm ./linux-4.19.320/drivers/gpu/
user@host:/tmp/t$ cp driver/product/kernel/include/linux/* ./linux-4.19.320/include/linux/
user@host:/tmp/t$ sed -i "/^obj-y/ s/$/ arm\//" linux-4.19.320/drivers/gpu/Makefile
user@host:/tmp/t$ sed -i '/.*drm\/Kconfig"/a source "drivers\/gpu\/arm\/Kconfig"' linux-4.19.320/drivers/video/Kconfig
user@host:/tmp/t$ cat <<- EOF >> ./linux-4.19.320/arch/x86/configs/x86_64_defconfig
CONFIG_MALI_MIDGARD=y
CONFIG_MALI_NO_MALI=y
CONFIG_MALI_DDK_VERSION=y
CONFIG_MALI_PLATFORM_NAME="devicetree"
CONFIG_MALI_EXPERT=y
CONFIG_MALI_DEBUG=y
CONFIG_MALI_FENCE_DEBUG=y
CONFIG_MALI_PRFCNT_SET_PRIMARY=y
CONFIG_MALI_GATOR_SUPPORT=n
CONFIG_MALI_MIDGARD_ENABLE_TRACE=n
CONFIG_MALI_SYSTEM_TRACE=n
CONFIG_MALI_KUTF=n
CONFIG_MALI_IRQ_LATENCY=n
CONFIG_CONFIG_MALI_CLK_RATE_TRACE_PORTAL=n
EOF
```

Add the driver directory to the Makefile, and add source to Kconfig for video.

Where to start in 2020 versions r0p0 - r29p0?

```
user@host:/tmp/t$ wget https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.19.320.tar.xz >/dev/null 2>&1
user@host:/tmp/t$ wget https://developer.arm.com/-/media/Files/downloads/mali-drivers/kernel/mali-bifrost-gpu/BX304L01B-SW-99002-r29p0-01eac0.tar > /dev/null 2>&1
user@host:/tmp/t$ tar xf linux-4.19.320.tar.xz
user@host:/tmp/t$ tar xf BX304L01B-SW-99002-r29p0-01eac0.tar
user@host:/tmp/t$ cp -r driver/product/kernel/drivers/gpu/arm ./linux-4.19.320/drivers/gpu/
user@host:/tmp/t$ cp driver/product/kernel/include/linux/* ./linux-4.19.320/include/linux/
user@host:/tmp/t$ sed -i "/^obj-y/ s/$/ arm\//" linux-4.19.320/drivers/gpu/Makefile
user@host:/tmp/t$ sed -i '/.*drm\/Kconfig"/a source "drivers\/gpu\/arm\/Kconfig"' linux-4.19.320/drivers/video/Kconfig
user@host:/tmp/t$ cat <<- EOF >> ./linux-4.19.320/arch/x86/configs/x86_64_defconfig
CONFIG_MALI_MIDGARD=y
CONFIG_MALI_NO_MALI=y
CONFIG_MALI_DDK_VERSION=y
CONFIG_MALI_PLATFORM_NAME="devicetree"
CONFIG_MALI_EXPERT=y
CONFIG_MALI_DEBUG=y
CONFIG_MALI_FENCE_DEBUG=y
CONFIG_MALI_PRFCNT_SET_PRIMARY=y
CONFIG_MALI_GATOR_SUPPORT=n
CONFIG_MALI_MIDGARD_ENABLE_TRACE=n
CONFIG_MALI_SYSTEM_TRACE=n
CONFIG_MALI_KUTF=n
CONFIG_MALI_IRQ_LATENCY=n
CONFIG_CONFIG_MALI_CLK_RATE_TRACE_PORTAL=n
EOF
```

Add the driver Kconfig parameters to the kernel defconfig for x86.

These parameters were stolen from Samsung device. Then removed all I though weren't relevant to emulation, and added NO_MALI.

Where to start in 2020 versions r0p0 - r29p0?

```
user@host:/tmp/t$ cd linux-4.19.320
user@host:/tmp/t/linux-4.19.320$ make defconfig
  HOSTCC  scripts/basic/fixdep
  HOSTCC  scripts/kconfig/conf.o
  YACC    scripts/kconfig/zconf.tab.c
  LEX     scripts/kconfig/zconf.lex.c
  HOSTCC  scripts/kconfig/zconf.tab.o
  HOSTLD  scripts/kconfig/conf
*** Default configuration is based on 'x86_64_defconfig'
#
# configuration written to .config
#
user@host:/tmp/t/linux-4.19.320$ make -j20 >/dev/null
drivers/gpu/arm/midgard/mali_kbase_core_linux.c:37:10: fatal error: mali_kbase_model_linux.h: No such file or directory
   37 | #include "mali_kbase_model_linux.h"
      |          ^~~~~~~~~~~~~~~~~~~~~~~~~~
compilation terminated.
make[4]: *** [scripts/Makefile.build:303: drivers/gpu/arm/midgard/mali_kbase_core_linux.o] Error 1
make[4]: *** Waiting for unfinished jobs....
make[3]: *** [scripts/Makefile.build:544: drivers/gpu/arm/midgard] Error 2
make[2]: *** [scripts/Makefile.build:544: drivers/gpu/arm] Error 2
make[2]: *** Waiting for unfinished jobs....
make[1]: *** [scripts/Makefile.build:544: drivers/gpu] Error 2
make: *** [Makefile:1086: drivers] Error 2
user@host:/tmp/t/linux-4.19.320$
```

Model is missing. Checked all older versions across midgard and bifrost. No luck in 2020.

How far can we get making our own model?

- Try to get it to compile, completing the code
  - Touch missing files, add missing functions and structs

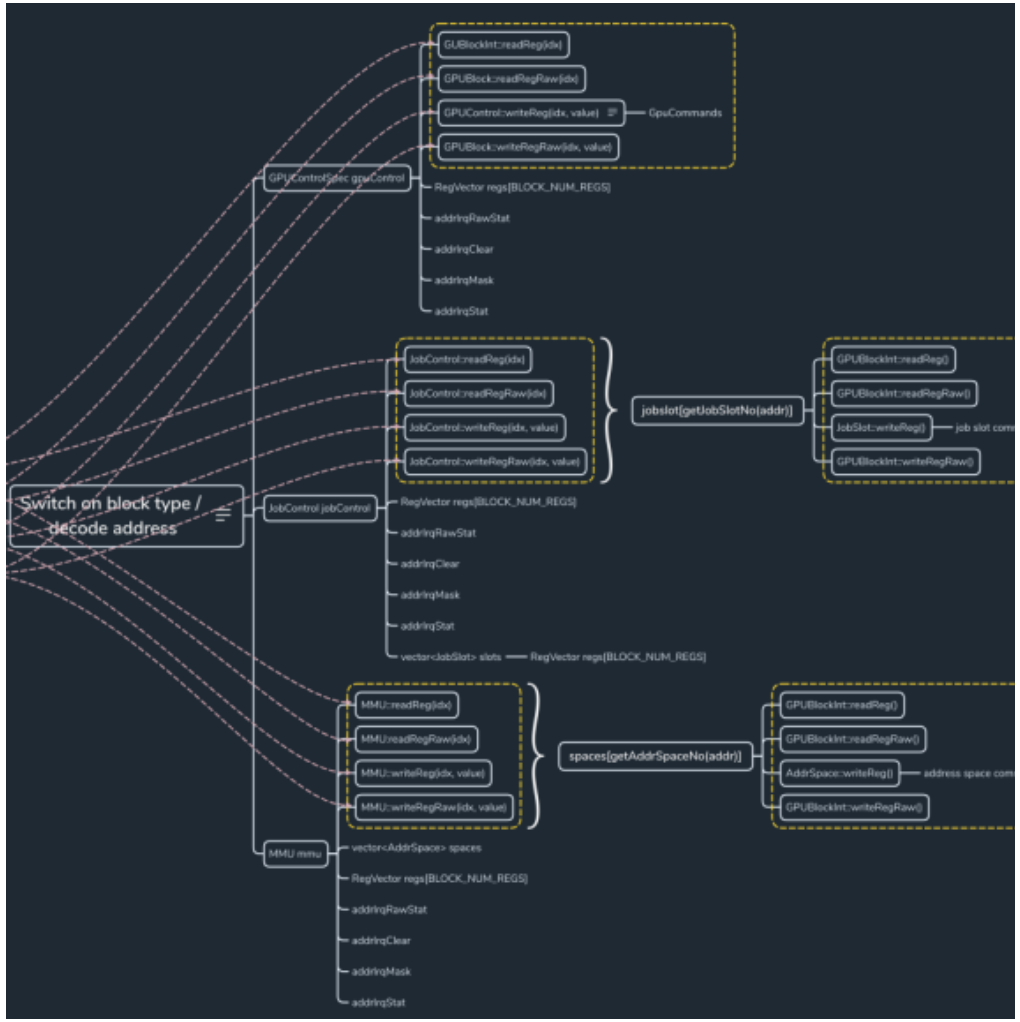
```
drivers/gpu/arm/midgard/mali_kbase_hwcnt_backend_jm.c: In function ‘kbasep_hwcnt_backend_jm_create’:
drivers/gpu/arm/midgard/mali_kbase_hwcnt_backend_jm.c:600:2: error: implicit declaration of function ‘gpu_model_set_dummy
_prfcnt_base_cpu’ [-Werror=implicit-function-declaration]
  600 |  gpu_model_set_dummy_prfcnt_base_cpu(backend->cpu_dump_va);
      |  ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
cc1: some warnings being treated as errors
make[4]: *** [scripts/Makefile.build:303: drivers/gpu/arm/midgard/mali_kbase_hwcnt_backend_jm.o] Error 1
make[4]: *** Waiting for unfinished jobs....
drivers/gpu/arm/midgard/device/backend/mali_kbase_device_jm.c:151:3: error: ‘kbase_gpu_device_create’ undeclared here (no
t in a function); did you mean ‘kbase_pm_device_data’?
  151 |  {kbase_gpu_device_create, kbase_gpu_device_destroy,
      |   ^~~~~~~~~~~~~~~~~~~~~~
      |   kbase_pm_device_data
drivers/gpu/arm/midgard/device/backend/mali_kbase_device_jm.c:151:28: error: ‘kbase_gpu_device_destroy’ undeclared here (
not in a function); did you mean ‘kbase_pm_device_data’?
  151 |  {kbase_gpu_device_create, kbase_gpu_device_destroy,
      |                            ^~~~~~~~~~~~~~~~~~~~~~~
      |                            kbase_pm_device_data
```

Several iterations later

- Compiler and linker guided code completion
  - Touch missing include files, add missing struct and function declarations
  - Linker informs which function definitions to add to c files
  - Need to extrapolate behavior from calling functions and struct usages
  - Not quite easy, I'd like to do less work
- Read the released model
  - https://github.com/ARM-software/nomali-model
    - C++ not compatible with driver code, assume written for Gem5

Read the released model

- Auditing using mindmaps
  - I like the idea of making software schematics similar to PCB layouts
  - Goal of code base is setting regmap
    - GPU Control
    - Job Control
    - MMU Control
  - Code modeled regs
    - irqRawStat, irqClear, irqMask, irqStat
- Modeled functions
  - regRead, regReadRaw
  - regWrite, regWriteRaw



Regmaps in the driver

- Memory mapped GPU registers for control, jobs, and mmu
  - driver/product/kernel/drivers/gpu/arm/midgard/gpu/mali_kbase_gpu_regmap.h

```
/* Job control registers */

#define JOB_CONTROL_BASE        0x1000

#define JOB_CONTROL_REG(r)      (JOB_CONTROL_BASE + (r))

#define JOB_IRQ_RAWSTAT         0x000   /* Raw interrupt s
#define JOB_IRQ_CLEAR           0x004   /* Interrupt clear
#define JOB_IRQ_MASK            0x008   /* Interrupt mask
#define JOB_IRQ_STATUS          0x00C   /* Interrupt statu
```

```
/* MMU control registers */

#define MEMORY_MANAGEMENT_BASE  0x2000
#define MMU_REG(r)              (MEMORY_MANAGEMENT_BASE +

#define MMU_IRQ_RAWSTAT         0x000   /* (RW) Raw interr
#define MMU_IRQ_CLEAR           0x004   /* (WO) Interrupt
#define MMU_IRQ_MASK            0x008   /* (RW) Interrupt
#define MMU_IRQ_STATUS          0x00C   /* (RO) Interrupt
```

```
/* GPU control registers */

#define GPU_CONTROL_BASE        0x0000
#define GPU_CONTROL_REG(r)      (GPU_CONTROL_BASE + (r))
#define GPU_ID                  0x000   /* (RO) GPU and re
#define L2_FEATURES             0x004   /* (RO) Level 2 ca
#define TILER_FEATURES          0x00C   /* (RO) Tiler Feat
#define MEM_FEATURES            0x010   /* (RO) Memory sys
#define MMU_FEATURES            0x014   /* (RO) MMU featur
#define AS_PRESENT              0x018   /* (RO) Address sp
#define GPU_IRQ_RAWSTAT         0x020   /* (RW) */
#define GPU_IRQ_CLEAR           0x024   /* (WO) */
#define GPU_IRQ_MASK            0x028   /* (RW) */
#define GPU_IRQ_STATUS          0x02C   /* (RO) */
```

- Released model matched the regmap
- Idea: add static mem mapped regs

My model

- Add static memory for registers
- Use driver macros to read values when called

```
#define REGISTER_SPACE_SIZE_DTS 0x5000
static u32 dummy_regs[REGISTER_SPACE_SIZE_DTS];

unsigned int kbase_reg_read(struct kbase_device *kbdev, u32 offset)
{
    if (offset == GPU_CONTROL_REG(GPU_IRQ_STATUS)) {
        return dummy_regs[offset] & dummy_regs[GPU_CONTROL_REG(GPU_IRQ_MASK)];
    }
    else if (offset == JOB_CONTROL_REG(JOB_IRQ_STATUS)) {
        return dummy_regs[offset] & dummy_regs[JOB_CONTROL_REG(JOB_IRQ_MASK)];
    }
    else if (offset == MMU_REG(MMU_IRQ_STATUS)) {
        return dummy_regs[offset] & dummy_regs[MMU_REG(MMU_IRQ_MASK)];
    }

    return dummy_regs[offset];
}
```

My model – how simple can I make it

- Hack: after every reg write, set all job slots to success

```
void kbase_reg_write(struct kbase_device *kbdev, u32 offset, u32 value) {
    dummy_regs[offset] = value;

    // just mark all jobs done successfully all the time
    dummy_regs[JOB_SLOT_REG(0, JS_STATUS)] = 0;
    dummy_regs[JOB_SLOT_REG(1, JS_STATUS)] = 0;
    dummy_regs[JOB_SLOT_REG(2, JS_STATUS)] = 0;
    dummy_regs[JOB_SLOT_REG(3, JS_STATUS)] = 0;
    dummy_regs[JOB_SLOT_REG(4, JS_STATUS)] = 0;
    dummy_regs[JOB_SLOT_REG(5, JS_STATUS)] = 0;
    dummy_regs[JOB_SLOT_REG(6, JS_STATUS)] = 0;
    dummy_regs[JOB_SLOT_REG(7, JS_STATUS)] = 0;
    dummy_regs[JOB_SLOT_REG(8, JS_STATUS)] = 0;
    dummy_regs[JOB_SLOT_REG(9, JS_STATUS)] = 0;
    dummy_regs[JOB_SLOT_REG(10, JS_STATUS)] = 0;
    dummy_regs[JOB_SLOT_REG(11, JS_STATUS)] = 0;
    dummy_regs[JOB_SLOT_REG(12, JS_STATUS)] = 0;
    dummy_regs[JOB_SLOT_REG(13, JS_STATUS)] = 0;
    dummy_regs[JOB_SLOT_REG(14, JS_STATUS)] = 0;
    dummy_regs[JOB_SLOT_REG(15, JS_STATUS)] = 0;
}
```

My model

- Compiles and links in tree build successfully
- Add initramfs (busybox) and proper qemu command
- Run and debug probe errors
  - Ret 0 in power mgmt init function

```
[    3.070321] mali mali.0: Kernel DDK version r29p0-01eac0
[    3.072360] mali mali.0: GPU identified as 0x0 arch 6.0.0 r0p1 status 0
[    3.075274] mali mali.0: No clock(s) available for rate tracing
[    3.579526] mali mali.0: Failed to soft-reset GPU (timed out after 500 ms), now attempting a hard reset
[    4.080364] mali mali.0: Failed to hard-reset the GPU (timed out after 500 ms)
[    4.082254] mali mali.0: Late backend initialization failed error = -22
[    4.083815] mali mali.0: Device initialization failed
[    4.084220] mali: probe of mali.0 failed with error -22
```

And it works surprisingly well

```
/tmp # printf 'A%.0s' `seq 100` > t
/tmp # xxd t
00000000: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00000010: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00000020: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00000030: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00000040: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00000050: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00000060: 4141 4141                                AAAA
/tmp # ./bug1 t
(INFO)[bug1.c:371] [+] opened file t O_RDONLY = 3
(INFO)[bug1.c:388] [+] mmaped file PROT_READ at = 0x7ffe9059e2e0
(INFO)[mali_utils.c:17] [+] opened mali device fd = 4
(INFO)[mali_utils.c:45] [+] mapped tracking page addr = 0x7ff3f59e900
(INFO)[mali_utils.c:73] [+] initialized mali version = K:r29p0-01eac0
(INFO)[mali_utils.c:111] [+] initialized mali jit mem
(INFO)[bug1.c:175] [+] mali imported user_buf = 0x7ff3f59ea000, to gp
(INFO)[bug1.c:134] [+] mali job submit ret 0
(INFO)[bug1.c:305] [+] mali ioctl soft event update ret >= 0
(INFO)[bug1.c:305] [+] mali ioctl soft event update ret >= 0
```

- CVE-2021-44828
  - Import and limited write to RO shared mappings
- Can write a pointer, a 0 or 1
- Exploitation on device may achieve pivot by targeting page caches of system libs

```
(INFO)[bug1.c:280] [+] mali job submit ret 0
/tmp # xxd t
00000000: 0000 f0ff ff7f 0000 0001 0001 0001 0001  ................
00000010: 0000 f0ff ff7f 0000 0001 0001 0001 0001  ................
00000020: 0000 f0ff ff7f 0000 0001 0001 0001 0001  ................
00000030: 0000 f0ff ff7f 0000 0001 0001 0001 0001  ................
00000040: 0000 f0ff ff7f 0000 0001 0001 0001 0001  ................
00000050: 0000 f0ff ff7f 0000 0001 0001 0001 0001  ................
00000060: 0001 4141                                ..AA
/tmp #
```

The ARM Mali Model

- In version r35p0 the mali driver started including their own working hardware model and has continued to update it
  - Probabilistic error generator in IRQs
  - Advanced modeling of Jobs and MMU actions
  - Support for both CSF and JM
    - Mid 2021 ARM including CSF support in version 28 and higher
    - CSF support slowly adopted by vendors over the last 3 years
    - CSF has added significant complexity and bugs
- I back-ported my model to version 5, and started using the included model

Can we improve on these models?

- Compile for ARM64
  - Need to augment DTB due to mandatory CONFIG_OF

```
# dump and disassemble dtb for arm virt machine
qemu-system-aarch64  -machine virt -machine dumpdtb=qemu.dtb
linux-4.19.135/scripts/dtc/dtc -I dtb ./qemu.dtb > qemu.dts

# modify the dts to add a mali gpu config
    gpu@c0000 {
      compatible = "arm,mali-midgard";
      reg = <0x0 0xc0000 0x0 0x400000>;
      interrupts = <0x0 0x6 0x4>, <0x0 0x7 0x4>, <0x0 0x8 0x4>;
      interrupt-names = "JOB", "MMU", "GPU";
      clock-names = "apb_pclk";
      clocks = <0x8000>;
    };

# recompile dtb from modified dts
linux-4.19.135/scripts/dtc/dtc -I dts -O dtb -o qemu_mod.dtb qemu.dts
```

Can we improve on these models?

- Compile out of tree to make modules
  - Need to provide build environment
  - Inspired by macros from Pixel kernel build system
- Model wasn’t maintained well through all versions
  - Lots of debugging and patching
- Release working models for versions 5 through 50
  - https://github.com/bernielampe1/mali_models

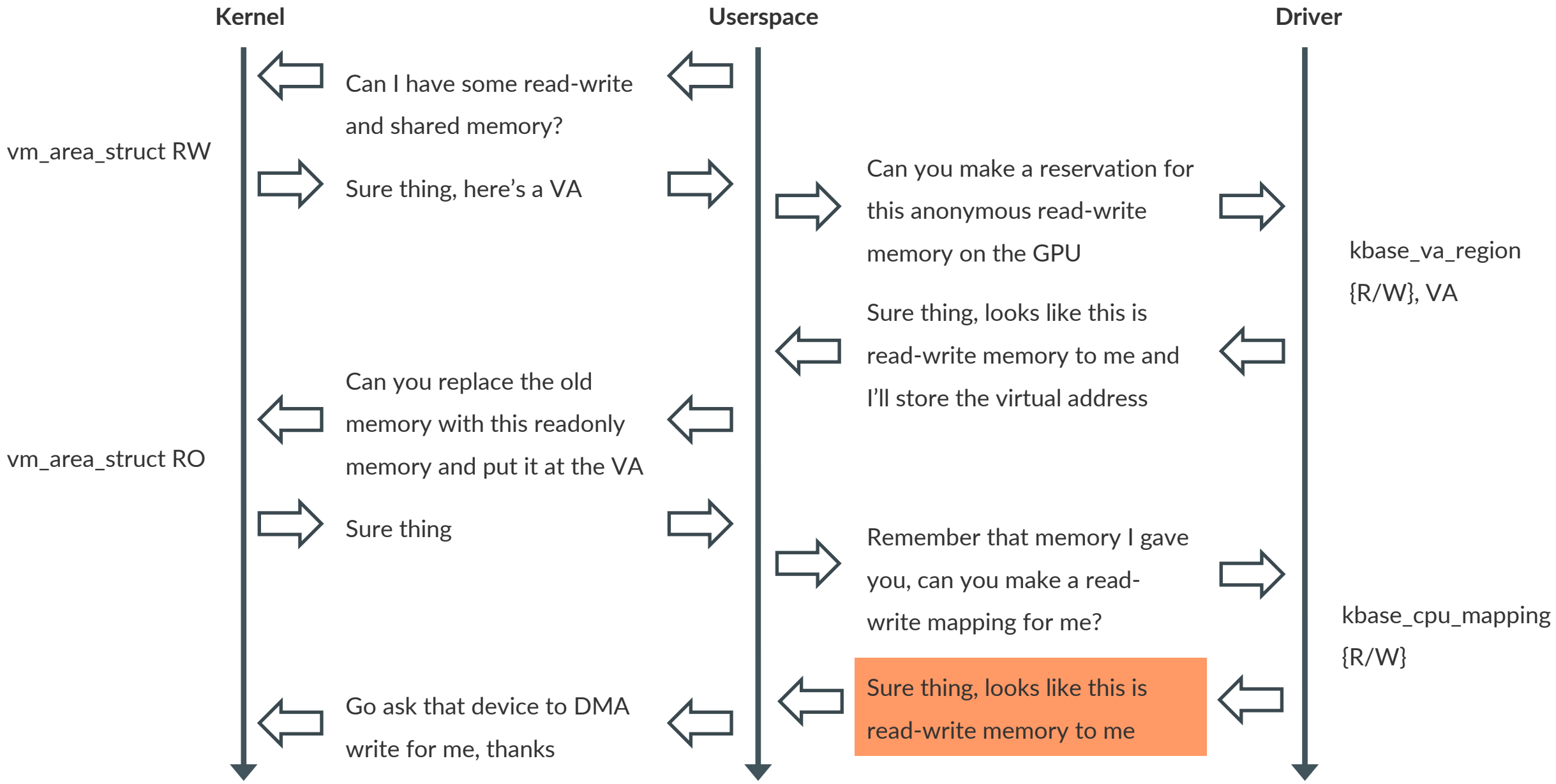
Use the Model to Test Bugs CVE-2022-22706

- Write to read only pages using DMA
- CPU write is set in the driver struct, but not validated when pinning
- Driver only considers GPU exports when using get_user_pages() interface

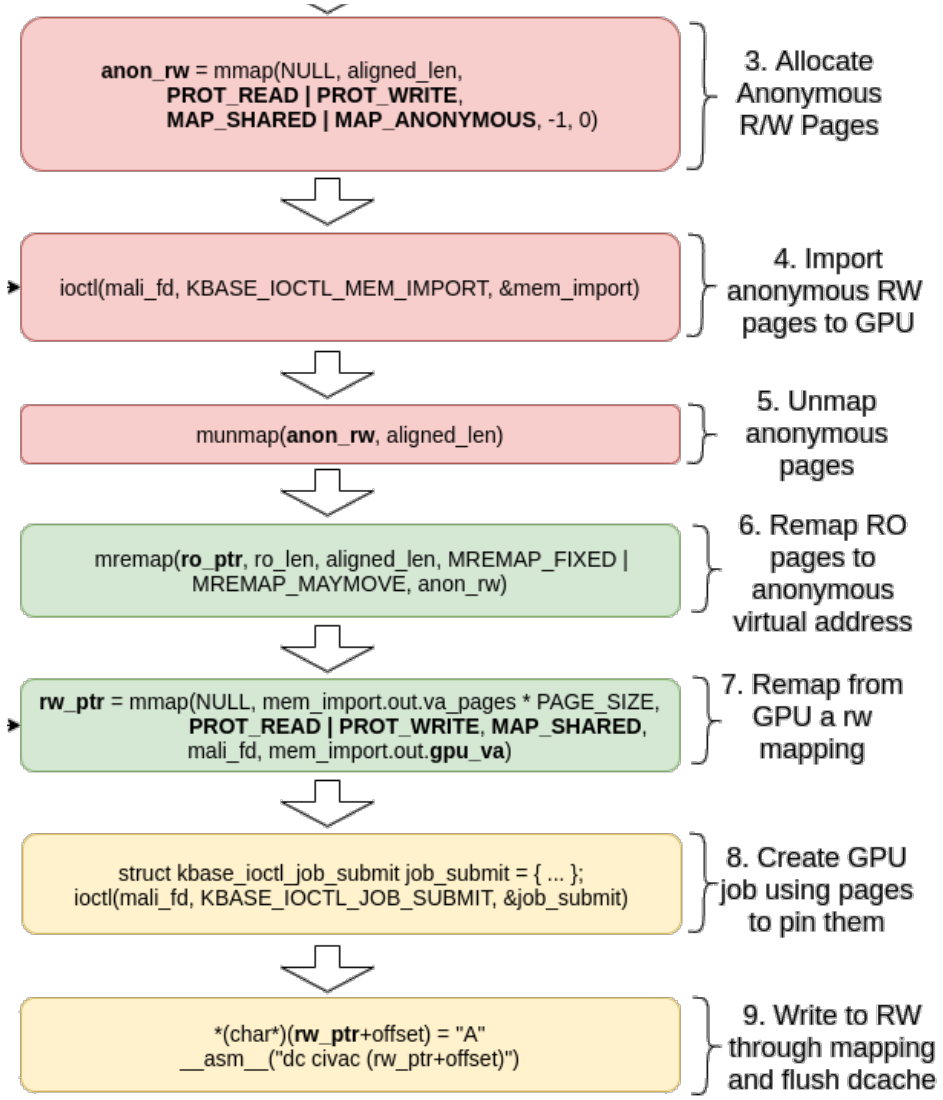
```
int kbase_jd_user_buf_pin_pages(struct kbase_context *kctx,
        struct kbase_va_region *reg)
{
    struct kbase_mem_phy_alloc *alloc = reg->gpu_alloc;
    struct page **pages = alloc->imported.user_buf.pages;
    unsigned long address = alloc->imported.user_buf.address;
    struct mm_struct *mm = alloc->imported.user_buf.mm;
    long pinned_pages;
    long i;
```

```
                pages, NULL);
#elif KERNEL_VERSION(5, 9, 0) > LINUX_VERSION_CODE
    pinned_pages = get_user_pages_remote(NULL, mm,
                address,
                alloc->imported.user_buf.nr_pages,
                reg->flags & KBASE_REG_GPU_WR ? FOLL_WRITE : 0,
                pages, NULL, NULL);
```

Use the Model to Test Bugs CVE-2022-22706



Primitive Userspace Calls CVE-2022-22706




```
/mnt # printf 'B%.0s' `seq 100` > t
/mnt # xxd t
00000000: 4242 4242 4242 4242 4242 4242 4242 4242  BBBBBBBBBBBBBBBB
00000010: 4242 4242 4242 4242 4242 4242 4242 4242  BBBBBBBBBBBBBBBB
00000020: 4242 4242 4242 4242 4242 4242 4242 4242  BBBBBBBBBBBBBBBB
00000030: 4242 4242 4242 4242 4242 4242 4242 4242  BBBBBBBBBBBBBBBB
00000040: 4242 4242 4242 4242 4242 4242 4242 4242  BBBBBBBBBBBBBBBB
00000050: 4242 4242 4242 4242 4242 4242 4242 4242  BBBBBBBBBBBBBBBB
00000060: 4242 4242                                BBBB
/mnt # ./bug2 -f t
(INFO)[utils.c:71] [+] opened file t O_RDONLY fd = 3
(INFO)[utils.c:80] [+] mmaped file PROT_READ at 0xffffb1708000
(INFO)[mali_utils.c:17] [+] opened mali device fd = 4
(INFO)[mali_utils.c:45] [+] mapped tracking page addr = 0xffffb1707000
(INFO)[mali_utils.c:73] [+] initialized mali version = K:r35p0-01eac0(GPL)
(INFO)[mali_utils.c:111] [+] initialized mali jit mem
(INFO)[bug2.c:147] [+] anon rw mapping made at 0xffffb1706000
(INFO)[bug2.c:166] [+] mali imported user_buf = 0xffffb1706000, to gpu_va = 0x41000
(INFO)[bug2.c:173] [+] munmaped anon_rw
(INFO)[bug2.c:180] [+] target RO vma at 0xffffb1708000 remapped to 0xffffb1706000
(INFO)[bug2.c:133] [+] mali job submit ret 0
/mnt # xxd t
00000000: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00000010: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00000020: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00000030: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00000040: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00000050: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00000060: 4141 4141                                AAAA
/mnt #
```

CVE-2022-36449 by Jann Horn

- UAF of physmem pages, mali will DMA
- Pulled POC from:
  - https://project-zero.issues.chromium.org/issues/42451459
- Added task_struct spray, compiled static for ARM64

```
/mnt # ./bug5
MEM_IMPORT result: flags = 0x400f, gpu_va=0x41000, va_pages=0x1
actual host+gpu VA: 0xffffae5a3000
gpu_va = 0x41000
  gpu_va is cookie, doing real mmap...
hexdump(0xffffae5a3000, 0x1000)
00000000  00 00 00 00 00 00 00 00 88 80 5d d7 ff ff 00 00  |..........].....|
00000010  00 00 00 00 00 00 00 00 a8 0d 49 00 00 00 00 00  |..........I.....|
00000020  d0 08 49 00 00 00 00 00 00 00 00 00 00 00 00 00  |..I.............|
00000030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
```

```
000005a0  02 00 00 00 00 00 00 00 48 89 5d d7 ff ff 00 00  |........H.].....|
000005b0  e8 85 5d d7 ff ff 00 00 10 27 00 00 00 00 00 00  |..]......'......|
000005c0  00 00 00 00 00 00 00 00 00 4f ab 9c e6 e7 3e b2  |.........O....>.|
000005d0  00 86 5d d7 ff ff 00 00 d8 0e 40 00 00 00 00 00  |..].......@.....|
000005e0  01 00 00 00 00 00 00 00 45 61 53 74 45 52 65 47  |........EaStEReG|
000005f0  67 5f 31 00 00 00 00 00 00 4f ab 9c e6 e7 3e b2  |g_1......O....>.|
00000600  80 87 5d d7 ff ff 00 00 a4 0f 40 00 00 00 00 00  |..].......@.....|
00000610  40 ce ee 09 00 00 00 00 0b 00 0b 00 00 00 00 00  |@...............|
```

How were these models made?

- Cut out irrelevant code
  - init code, power management code, timers, and IRQ handling
- Create null stub functions
  - Easier in some cases to have a null implementation
- Create static resources for IRQ and iomem
  - ioremaps all point to static buffers
- Create functions to emulate needed hardware invocations
  - Status registers are most important for getting driver to run

Limitations of this Approach

- Can’t emulated complicated IRQ or MMU interactions
  - Imagine bugs on the IRQ path which can be influenced from userspace
  - Usually these paths are a small portion of the driver and can be audited

```
void kbase_mmu_page_fault_worker(struct work_struct *data)
{
    u64 fault_pfn;
    u32 fault_status;
    size_t new_pages;
    size_t fault_rel_pfn;
    struct kbase_as *faulting_as;
    int as_no;
    struct kbase_context *kctx;
```

```
    kbase_gpu_vm_unlock(kctx);

    /* If the memory pool was insufficient then grow it and retry.
     * Otherwise fail the allocation.
     */
    if (pages_to_grow > 0) {
            struct kbase_mem_pool *const mem_pool =
                &kctx->mem_pools.small[region->gpu_alloc->group_id];

            ret = kbase_mem_pool_grow(mem_pool, pages_to_grow);
    }
```

Why is this possible?

```c
static int __init kbase_driver_init(void)
{
    int ret;

    ret = kbase_platform_register();
    if (ret)
        return ret;

    ret = platform_driver_register(&kbase_platform_driver);

    if (ret)
        kbase_platform_unregister();

    return ret;
}
module_init(kbase_driver_init);
```

- Jonathan Corbet, 2011, LWN, “The platform device API”
- Platform devices are not discoverable like PCI

Not everybody in the kernel community is enamored with platform devices; they seem like a bit of a hack used to encode information about specific hardware platforms into the kernel. Additionally, the platform data mechanism lacks any sort of type checking; drivers must simply assume that they have been passed a structure of the expected type. Even so, platform devices are heavily used, and

https://lwn.net/Articles/448499/

Can we use these lessons from mali?

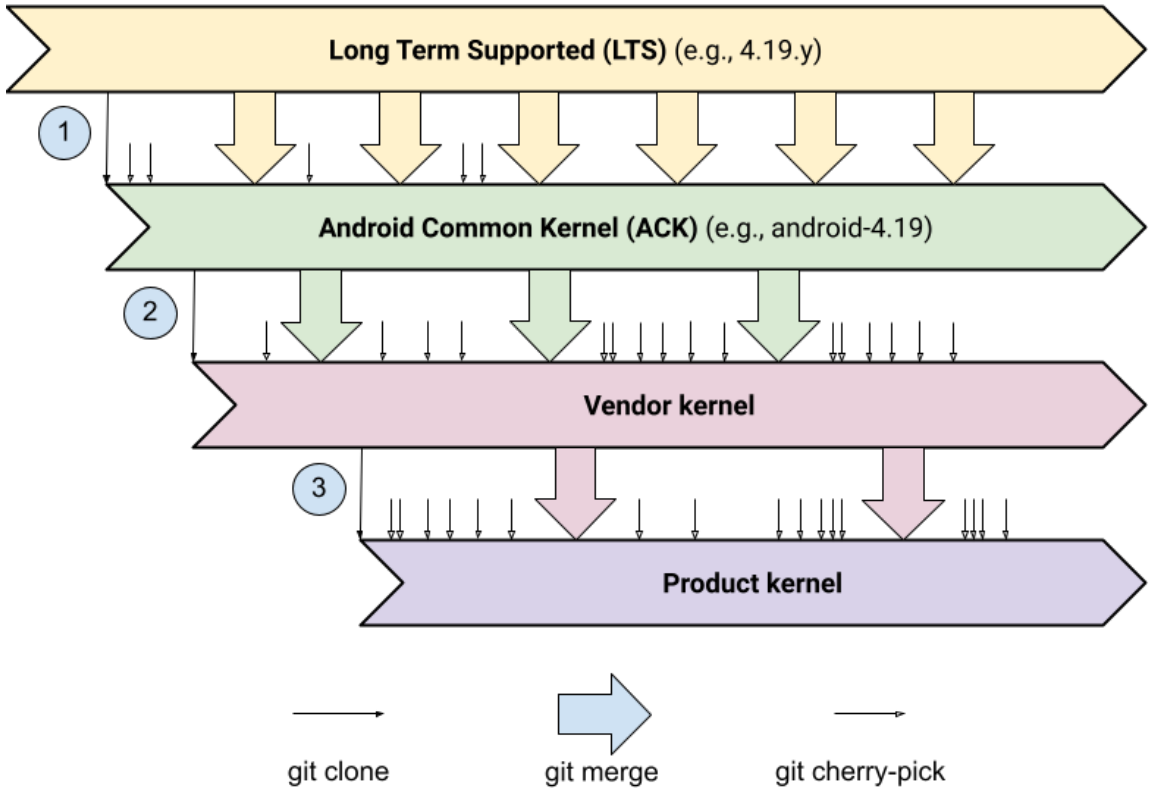
- Should be able to:
  - Cut out irrelevant code
  - Create null stub functions
  - Create static memory resources for IPC
  - Create functions to emulate certain hardware actions
  - Add DTB entries
  - Add build and emulation env
- Try to apply to Qualcomm MSM kernel NPU driver
  - New problems arose
  - Found symbols not included in Linux kernel or driver?

Issue encountered with Qualcomm

- ARM wants wide adoption of their GPU
  - TVs, game consoles, cars
  - Readily integrates into Linux LTS kernels
- Qualcomm controls the hardware, kernel, and driver stack and packages them together
  - No external pressure to version their code for releasing
  - Augment their kernel and drivers
- Generic kernel image (GKI)
  - Isn’t the GKI supposed to be…generic?

What is the GKI really?

- Before 2020, kernels were more fragmented
  - Android / Google kernels *had* a bad fragmentation / security reputation
- GKI unifies core kernel and moves SoC and BSP code into loadable modules
- Maintains a stable and tested KMI
  - Don’t break userspace
  - Enable stable kernel with security update mechanism



Kernel Module Interface

It's critical to maintain a stable kernel module interface (KMI) for vendor modules. The GKI kernel is built and shipped in binary form and vendor-loadable modules are built in a separate tree. The resulting GKI kernel and vendor modules must work as though they were built together.

- After the KMI branch is frozen, changes are allowed but can't break the KMI. These changes include the following:
  - Config changes
  - Kernel code changes

https://source.android.com/docs/core/architecture/kernel/stable-kmi

The Generic Kernel Image (GKI) reduces kernel fragmentation by aligning closely with the upstream Linux kernel. However, there are valid reasons why some patches can't be accepted upstream, and there are product schedules that must be met, so some patches are maintained in the Android Common Kernel (ACK) sources from which the GKI is built.

https://source.android.com/docs/core/architecture/kernel/kernel-code

How does one change the GKI?

- ACK can add new data to structs before and after KMI freezing

```
/*
 * Macros to use _before_ the ABI is frozen
 */

/*
 * ANDROID_KABI_RESERVE
 *   Reserve some "padding" in a structure for potential future use.
 *   This normally placed at the end of a structure.
 *   number: the "number" of the padding variable in the structure.  Start with
 *   1 and go up.
 */
#ifdef CONFIG_ANDROID_KABI_RESERVE
#define ANDROID_KABI_RESERVE(number)    _ANDROID_
#else
#define ANDROID_KABI_RESERVE(number)
#endif
```

```
/*
 * Macros to use _after_ the ABI is frozen
 */

/*
 * ANDROID_KABI_USE(number, _new)
 *   Use a previous padding entry that was defined with ANDROID_KABI_RESERVE
 *   number: the previous "number" of the padding variable
 *   _new: the variable to use now instead of the padding variable
 */
#define ANDROID_KABI_USE(number, _new)      \
    _ANDROID_KABI_REPLACE(_ANDROID_KABI_RESERVE(number), _new)
```

How does one change the GKI?

- Vendors and OEMs can add new structs if reserved

```
/*
 * ANDROID_VENDOR_DATA
 *   Reserve some "padding" in a structure for potential future use.
 *   This normally placed at the end of a structure.
 *   number: the "number" of the padding variable in the structure.  Start with
 *   1 and go up.
 *
 * ANDROID_VENDOR_DATA_ARRAY
 *   Same as ANDROID_VENDOR_DATA but allocates an array of u64 with
 *   the specified size
 */
#ifdef CONFIG_ANDROID_VENDOR_OEM_DATA
#define ANDROID_VENDOR_DATA(n)      u64 android_vendor_data##n
#define ANDROID_VENDOR_DATA_ARRAY(n, s) u64 android_vendor_data##n[s]

#define ANDROID_OEM_DATA(n)     u64 android_oem_data##n
#define ANDROID_OEM_DATA_ARRAY(n, s)    u64 android_oem_data##n[s]
```

How does one change the GKI?

- Add kernel code to support drivers
  - Add symbols to the android/abi_gki_aarch64.stg file
  - Push to AOSP for review, usually released in new GKI in a month

Extend the KMI

While KMI symbols and related structures are maintained as stable (meaning changes that break stable interfaces in a kernel with a frozen KMI cannot be accepted) the GKI kernel remains open to extensions so that devices shipping later in the year don't need to define all their dependencies before the KMI is frozen. To extend the KMI, you can add new symbols to the KMI for new or existing exported kernel functions, even if the KMI is frozen. New kernel patches might also be accepted if they don't break the KMI.

https://source.android.com/docs/core/architecture/kernel/abi-monitor

Where does this leave the GKI?

- Move all SoC and OEM code to loadable modules
- Document and enforce a stable KMI per release
  - AOSP, Vendors, and OEMs can reserve struct space
  - Anyone supporting a module can submit kernel code changes for review
- Does this really reduce fragmentation and improve security?
  - Allows for more orderly mechanism for updates
  - Fragmentation is documented and managed

```
user@host:~/kernels/pixel/aosp/android$ ls
abi_gki_aarch64              abi_gki_aarch64_honda    abi_gki_aarch64_pasa       abi_gki_aarch64_unisoc
abi_gki_aarch64_asr          abi_gki_aarch64_honor    abi_gki_aarch64_pixel      abi_gki_aarch64_virtual_device
abi_gki_aarch64_asus         abi_gki_aarch64_imx      abi_gki_aarch64_qcom       abi_gki_aarch64_vivo
abi_gki_aarch64_db845c       abi_gki_aarch64_meizu    abi_gki_aarch64_rockchip   abi_gki_aarch64_xiaomi
abi_gki_aarch64_exynos       abi_gki_aarch64_moto     abi_gki_aarch64_sony       abi_gki_protected_exports_aarch64
abi_gki_aarch64_exynosauto   abi_gki_aarch64_mtk      abi_gki_aarch64.stg        abi_gki_protected_exports_x86_64
abi_gki_aarch64_fips140      abi_gki_aarch64_nothing  abi_gki_aarch64_transsion  gki_aarch64_protected_modules
abi_gki_aarch64_galaxy       abi_gki_aarch64_oplus    abi_gki_aarch64_tuxera     gki_x86_64_protected_modules
user@host:~/kernels/pixel/aosp/android$
```

Building an NPU Model

- Run the formula, cutout irq/pm, mock up, dtb, add ioregs, etc
- Here I hijacked the devm_ioremap() call and added special cases
- Had to copy in many Qualcomm GKI changes for their KMI

```
#define REG_SIZE_TCM_DTS (0x20 * PAGE_SIZE)
#define REG_SIZE_CORE_DTS (0x10 * PAGE_SIZE)
#define REG_SIZE_CC_DTS (0x10 * PAGE_SIZE)
#define REG_SIZE_APSS_SHARED_DTS (0x10 * PAGE_SIZE)
#define REG_SIZE_TCSR_DTS (0x40 * PAGE_SIZE)
#define REG_SIZE_QFPROM_DTS (7 * PAGE_SIZE)

static u8 dummy_tcm_regs[REG_SIZE_TCM_DTS];
static u8 dummy_core_regs[REG_SIZE_CORE_DTS];
static u8 dummy_cc_regs[REG_SIZE_CC_DTS];
static u8 dummy_apss_shared_regs[REG_SIZE_APSS_SHARED_DTS];
static u8 dummy_tcsr_regs[REG_SIZE_TCSR_DTS];
static u8 dummy_qfprom_physical[REG_SIZE_QFPROM_DTS];

static int npu_host_ctrl_status_ipc_addr_ready = 0;
```

```
uint32_t npu_core_reg_read(struct npu_device *npu_dev, uint32_t off)
{
    if (off == REG_NPU_FW_CTRL_STATUS) {
        uint32_t ret = FW_CTRL_STATUS_MAIN_THREAD_READY_VAL | FW_CTRL_STATUS_PWR_NOTIFY_DONE_VAL;

        // have the ipc queues been init'd?
        if (npu_host_ctrl_status_ipc_addr_ready) {
            ret |= FW_CTRL_STATUS_LOG_READY_VAL;
        }
        return ret;
    }

    return npu_reg_read(npu_dev->core_io.base, npu_dev->core_io.size, off);
}

void npu_core_reg_write(struct npu_device *npu_dev, uint32_t off, uint32_t val)

    (val == HOST_CTRL_STATUS_IPC_ADDRESS_READY_VAL) {
     npu_host_ctrl_status_ipc_addr_ready = 1; // mark global for ipc queues init'd

_reg_write(npu_dev->core_io.base, npu_dev->core_io.size, off, val);
```

```
    npu_dev->core_io.phy_addr = res->start;
#if !IS_ENABLED(CONFIG_NO_NPU)
    npu_dev->core_io.base = devm_ioremap(&pdev->dev, res->start,
                npu_dev->core_io.size);
#else
    npu_dev->core_io.base = (void*)dummy_core_regs;
#endif
    if (unlikely(!npu_dev->core_io.base)) {
```

Building an NPU Model

- Works well, not polished yet
- Turned on high level debugging for detailed kernel messages
- Created a tester for all ioctls

```
/ # ./load_mod.sh ./msm_npu.ko
rmmod: remove 'msm_npu': No such file or directory
[   18.903829] NPU_DBG: npu_probe: 2455 core phy address=0x99f0000 virt=(____ptrval____)
[   18.904066] NPU_DBG: npu_probe: 2478 tcm phy address=0x9900000 virt=(____ptrval____)
[   18.904224] NPU_DBG: npu_probe: 2501 cc_io phy address=0x9980000 virt=(____ptrval____)
[   18.904410] NPU_DBG: npu_probe: 2524 tcsr phy address=0x1f40000 virt=(____ptrval____)
[   18.904624] NPU_DBG: npu_probe: 2547 apss_shared phy address=0x17c00000 virt=(____ptrval____)
[   18.904866] NPU_DBG: npu_probe: 2568 qfprom_physical phy address=0x780000 virt=(____ptrval____)
[   18.905509] msm_npu 9900000.qcom,msm_npu: 9900000.qcom,msm_npu supply vdd not found, using dummy regulator
[   18.906836] msm_npu 9900000.qcom,msm_npu: Linked as a consumer to regulator.0
[   18.907184] msm_npu 9900000.qcom,msm_npu: 9900000.qcom,msm_npu supply vdd_cx not found, using dummy regulator
[   18.908008] NPU_INFO: npu_parse_dt_bw: 1860 NPU BW client sets up successfully
[   18.908271] NPU_DBG: npu_enable_core_power: 602 Enable core power 0
[   18.908615] NPU_DBG: npu_enable_clocks: 931 Notify cdsprm 4
[   18.909325] NPU_DBG: npu_hw_info_init: 2405 NPU_HW_VERSION 0x0
[   18.909519] NPU_DBG: npu_disable_core_power: 634 Disable core power 1
[   18.909730] NPU_DBG: npu_disable_clocks: 1004 Notify cdsprm clock off
[   18.910098] NPU_DBG: npu_disable_core_power: 650 setting back to power level=0
[   18.910624] NPU_DBG: npu_of_parse_pwrlevels: 2021 clk apb_pclk rate [9600000]:[24000000]
[   18.910907] NPU_DBG: npu_of_parse_pwrlevels: 2021 clk apb_pclk rate [300000000]:[24000000]
[   18.911131] NPU_DBG: npu_of_parse_pwrlevels: 2021 clk apb_pclk rate [350000000]:[24000000]
[   18.911354] NPU_DBG: npu_of_parse_pwrlevels: 2021 clk apb_pclk rate [400000000]:[24000000]
[   18.911674] NPU_DBG: npu_of_parse_pwrlevels: 2021 clk apb_pclk rate [600000000]:[24000000]
[   18.911926] NPU_DBG: npu_of_parse_pwrlevels: 2021 clk apb_pclk rate [715000000]:[24000000]
[   18.912221] NPU_WARN: npu_adjust_max_power_level: 1923 can't find clock cal_hm0_clk
[   18.912442] NPU_DBG: npu_of_parse_pwrlevels: 2031 initial-pwrlevel 4
[   18.912643] NPU_DBG: npu_of_parse_pwrlevels: 2043 init power level 4 max 5 min 0
[   18.912890] NPU_INFO: npu_pwrctrl_init: 2080 npubw-dev-names are not defined
[   18.914115] NPU_DBG: npu_probe: 2636 drvdata (____ptrval____) (____ptrval____)
[   18.916919] NPU_DBG: npu_set_cur_state: 1072 request state=0
[   18.917314] NPU_DBG: npu_calc_power_level: 714 therm=5 active=4 uc=5 set level=5
[   18.917505] NPU_DBG: npu_set_power_level: 733 power is not enabled during set request
# Add the following lines to the .gdbinit or copy into kernel gdb session
add-symbol-file ${PWD}/msm_npu.ko 0xffff000000b50000 -s .bss 0xffff000000b63540 -s .data 0xffff000000b63000
directory ${KERNEL_SRC}
/ # ls -l /dev/msm_npu
crw-rw-r--    1 0        0         238,   0 Sep  3 17:51 /dev/msm_npu
/ #
```

Why are there *still* bugs in drivers?

- Linux developers can’t take responsibility
- Product vendors like Samsung integrate drivers as users
- Peripheral developers have both hw and sw teams
  - Priorities are userspace features and hardware functionality
  - Driver testing is a secondary priority
- Iron triangle of product management
  - Better, faster, cheaper
    - Microsoft - Tying security to employee performance
    - AI Co-pilots

What to do about it

- Rust in the kernel will save us. How's that going?
  - 9/2/2024 - "Rust Linux kernel maintainer steps down"
  - https://www.theregister.com/2024/09/02/rust_for_linux_maintainer_steps_down/
- Get driver developers to release their pre-silicon testing environments to research community.
  - Bug bounty programs have been waning
- Build a lot of driver models

Conclusion

- Vulnerability discovery is an active process
  - Recommend building a lot of hacky personal tools
- Bar of Linux driver VR can be lowered with less effort
  - Cut up, stub out, add static memory, guess, debug, repeat
- POC out theories and use it to iteratively learn more
  - Development of emulators will increase understanding along the way

Questions