AUGUST 9-10, 2023

BRIEFINGS

Make KSMA Great Again: The Art of Rooting Android devices by GPU MMU features

WANG, YONG (@ThomasKing2014)

Alibaba Cloud Pandora Lab

Whoami

- WANG, YONG @ThomasKing2014@infosec.exchange
  - @ThomasKing2014 on Twitter/Weibo
- Security Engineer of Alibaba Cloud
- Focus on Android/Chrome vulnerability
- Speaker at BlackHat{ASIA/EU/USA}/HITBAMS/Zer0Con/POC/CanSecWest
- Nominated at Pwnie Award 2019(Best Privilege Escalation)

Agenda

- Introduction
- GPU version of the KSMA exploitation technique
- Case study
- Conclusion

Linux kernel VMM 101

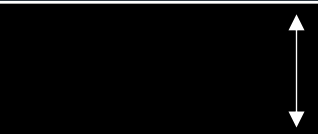
- Process isolation is a set of different hardware and software technologies designed to protect each process from other processes on the operating system. – Wikipedia
- `cat /proc/<pid>/maps`

```
7cdc448000-7cdc5a6000 ---p 00000000 00:00 0
7cdc5a6000-7cdc5a7000 ---p 00000000 00:00 0
7cdc5a7000-7cdc5af000 rw-p 00000000 00:00 0                              [anon:thread signal stack]
7cdc5af000-7cdc5b0000 rw-p 00000000 00:00 0                              [anon:arc4random data]
7cdc5b0000-7cdc5b2000 rw-p 00000000 00:00 0                              [anon:ReadFileToBuffer]
7cdc5b2000-7cdc5b3000 rw-p 00000000 00:00 0                              [anon:arc4random data]
7cdc5b3000-7cdc5b5000 r--p 00000000 00:00 0                              [vvar]
7cdc5b5000-7cdc5b6000 r-xp 00000000 00:00 0                              [vdso]
7cdc5b6000-7cdc5ee000 r--p 00000000 07:60 16                             /apex/com.android.runtime/bin/linker64
7cdc5ee000-7cdc6d8000 r-xp 00038000 07:60 16                             /apex/com.android.runtime/bin/linker64
7cdc6d8000-7cdc6e0000 r--p 00122000 07:60 16                             /apex/com.android.runtime/bin/linker64
7cdc6e0000-7cdc6e2000 rw-p 00129000 07:60 16                             /apex/com.android.runtime/bin/linker64
7cdc6e2000-7cdc6eb000 rw-p 00000000 00:00 0                              [anon:.bss]
7cdc6eb000-7cdc6ec000 r--p 00000000 00:00 0                              [anon:.bss]
7cdc6ec000-7cdc6ee000 rw-p 00000000 00:00 0                              [anon:.bss]
7fed13d000-7fed13e000 ---p 00000000 00:00 0
7fed13e000-7fed93d000 rw-p 00000000 00:00 0                              [stack]
```

Linux kernel VMM 101

struct mm_struct

```
7cdc448000-7cdc5a6000 ---p 00000000 00:00 0
7cdc5a6000-7cdc5a7000 ---p 00000000 00:00 0
7cdc5a7000-7cdc5af000 rw-p 00000000 00:00 0                              [anon:thread signal stack]
7cdc5af000-7cdc5b0000 rw-p 00000000 00:00 0                              [anon:arc4random data]
7cdc5b0000-7cdc5b2000 rw-p 00000000 00:00 0                              [anon:ReadFileToBuffer]
7cdc5b2000-7cdc5b3000 rw-p 00000000 00:00 0                              [anon:arc4random data]
7cdc5b3000-7cdc5b5000 r--p 00000000 00:00 0                              [vvar]
7cdc5b5000-7cdc5b6000 r-xp 00000000 00:00 0                              [vdso]
7cdc5b6000-7cdc5ee000 r--p 00000000 07:60 16                             /apex/com.android.runtime/bin/linker64
7cdc5ee000-7cdc6d8000 r-xp 00038000 07:60 16                             /apex/com.android.runtime/bin/linker64
7cdc6d8000-7cdc6e0000 r--p 00122000 07:60 16                             /apex/com.android.runtime/bin/linker64
7cdc6e0000-7cdc6e2000 rw-p 00129000 07:60 16                             /apex/com.android.runtime/bin/linker64
7cdc6e2000-7cdc6eb000 rw-p 00000000 00:00 0                              [anon:.bss]
7cdc6eb000-7cdc6ec000 r--p 00000000 00:00 0                              [anon:.bss]
7cdc6ec000-7cdc6ee000 rw-p 00000000 00:00 0                              [anon:.bss]
7fed13d000-7fed13e000 ---p 00000000 00:00 0
7fed13e000-7fed93d000 rw-p 00000000 00:00 0                              [stack]
```



struct vm_area_struct

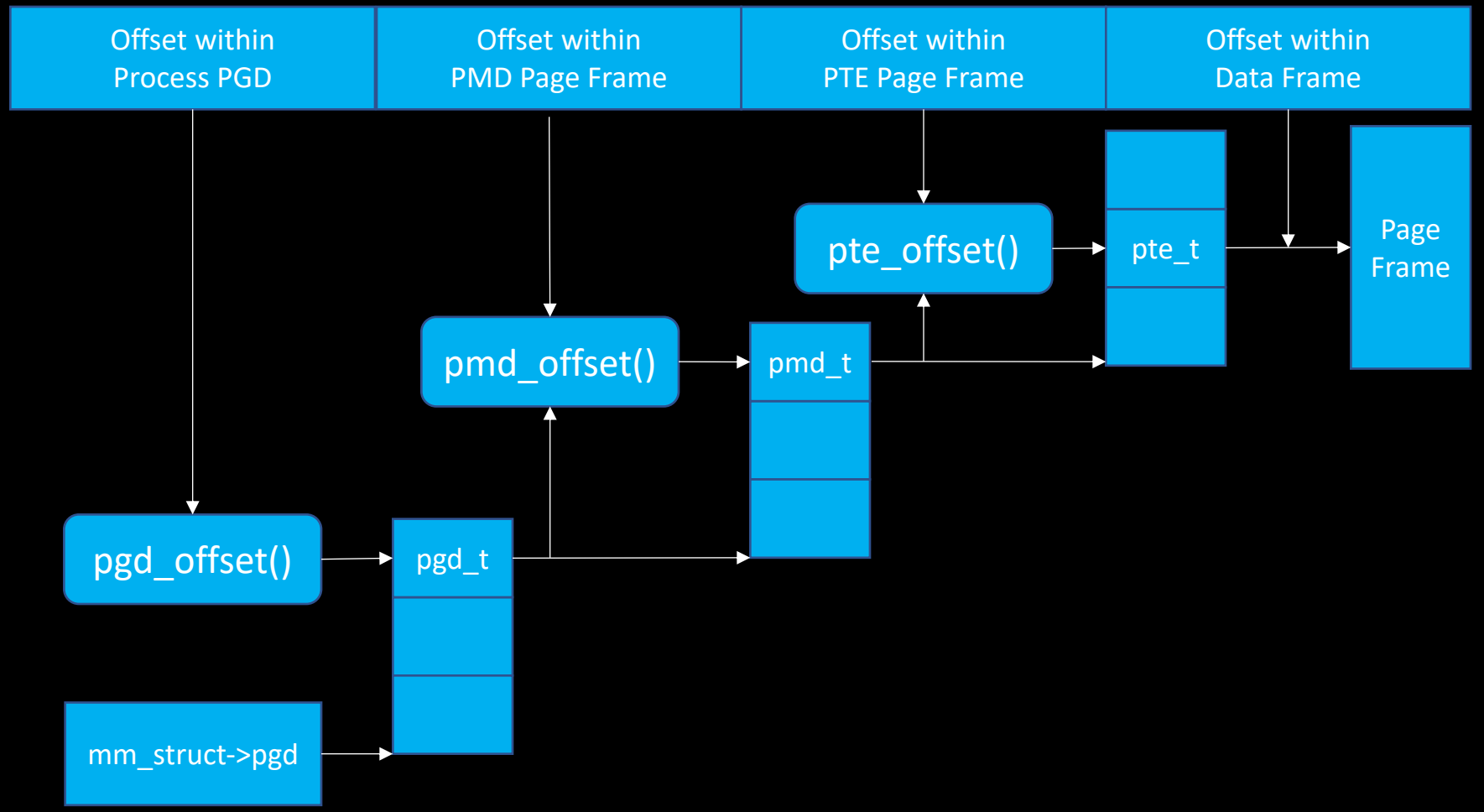
Linux kernel VMM 101

```
struct mm_struct {
        struct vm_area_struct *mmap;                    /* list of VMAs */
        struct rb_root mm_rb;
        …
        unsigned long mmap_base;    /* base of mmap area */
        …
         pgd_t * pgd;
        …
        /* store ref to file /proc/<pid>/exe symlink points to */
        struct file __rcu *exe_file;
```

Linux kernel VMM 101

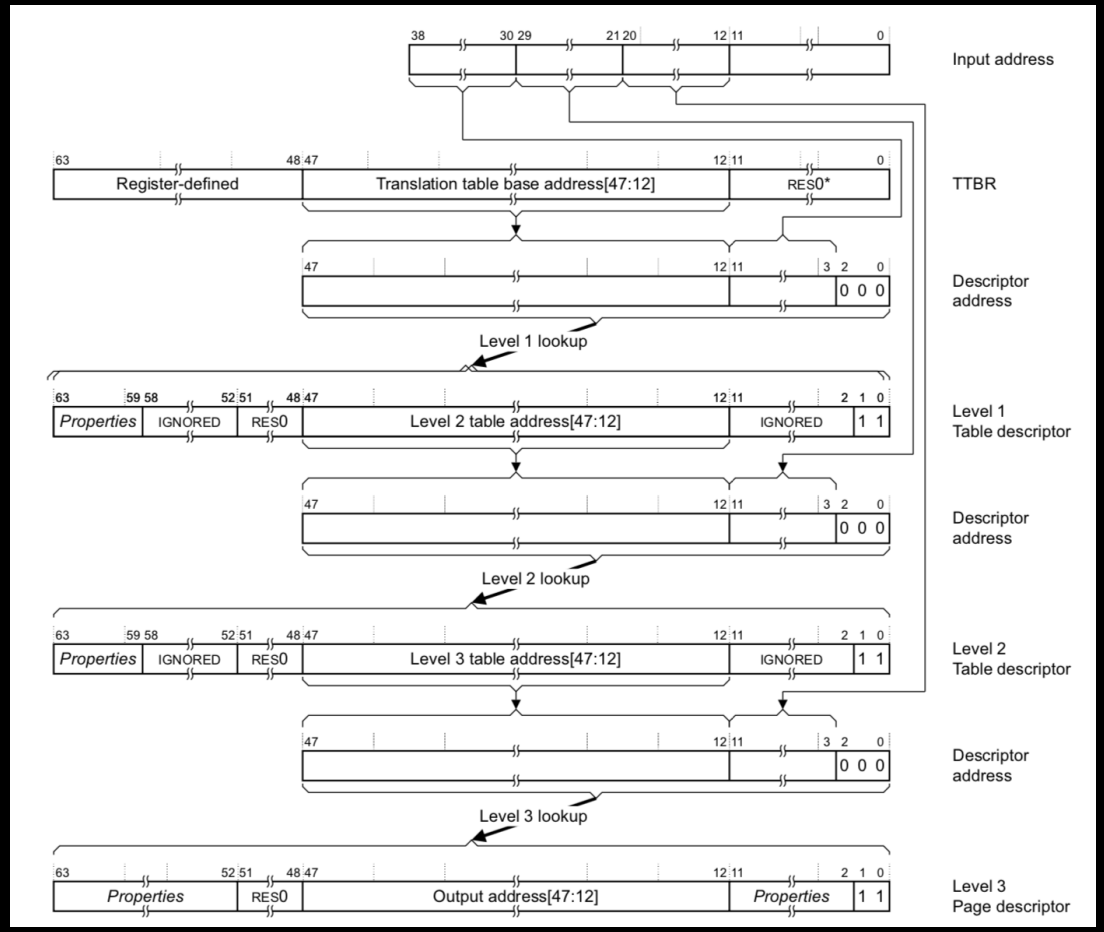
```
struct vm_area_struct {
        unsigned long vm_start;
        unsigned long vm_end;
        struct rb_node vm_rb;
        unsigned long vm_flags;
        const struct vm_operations_struct *vm_ops;
        unsigned long vm_pgoff;
        struct file * vm_file;
        …
```

Linux kernel VMM 101



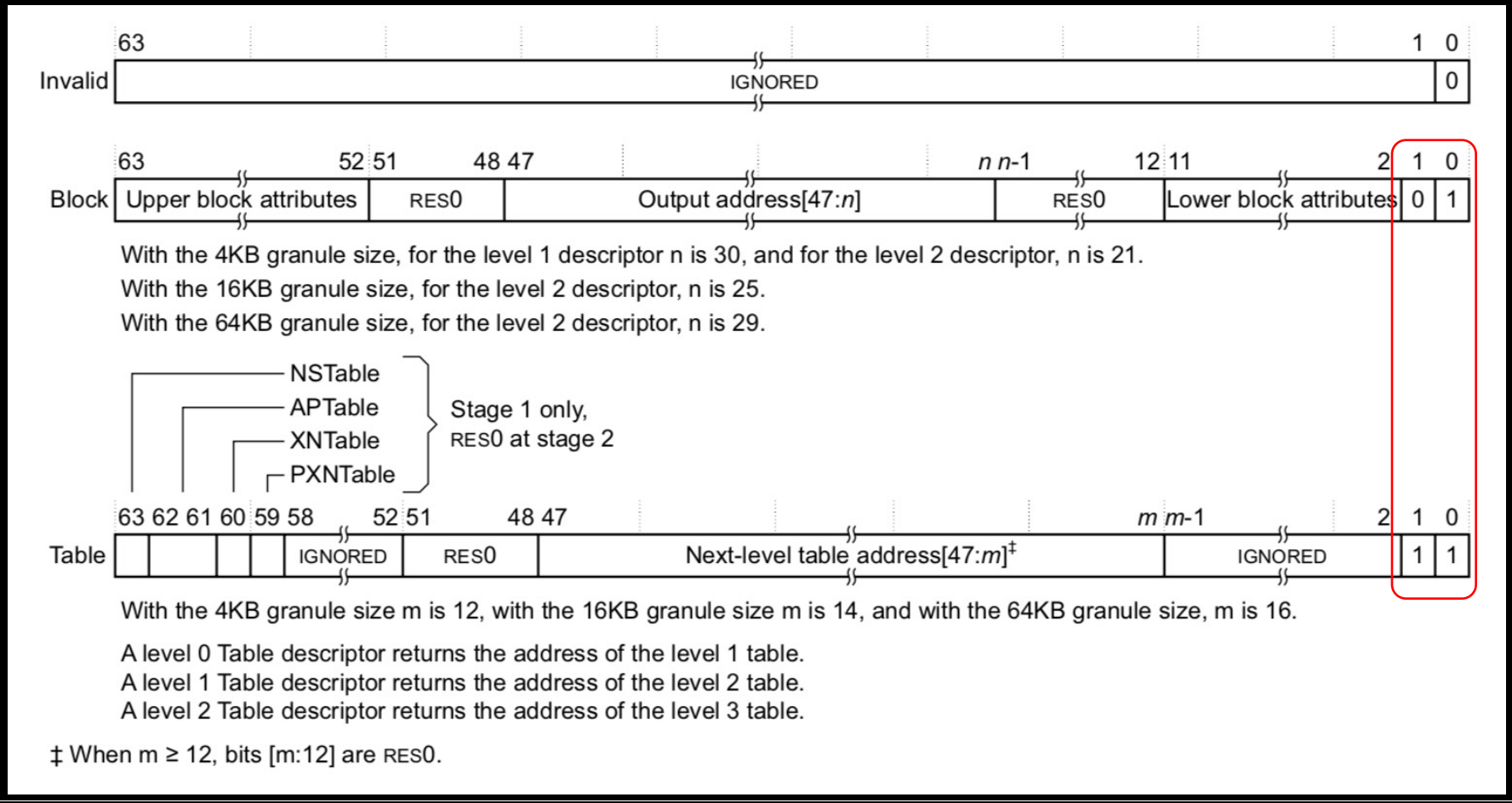
Linux kernel VMM 101

- For Android
  - 4KB granule
  - 39-bit (512GB)
  - Three levels
- TTBRx
  - TTBR0 - user address
    - Up to 0x0000_007F_FFFF_FFFF
  - TTBR1 - kernel address
    - Start from 0xFFFF_FF80_0000_0000



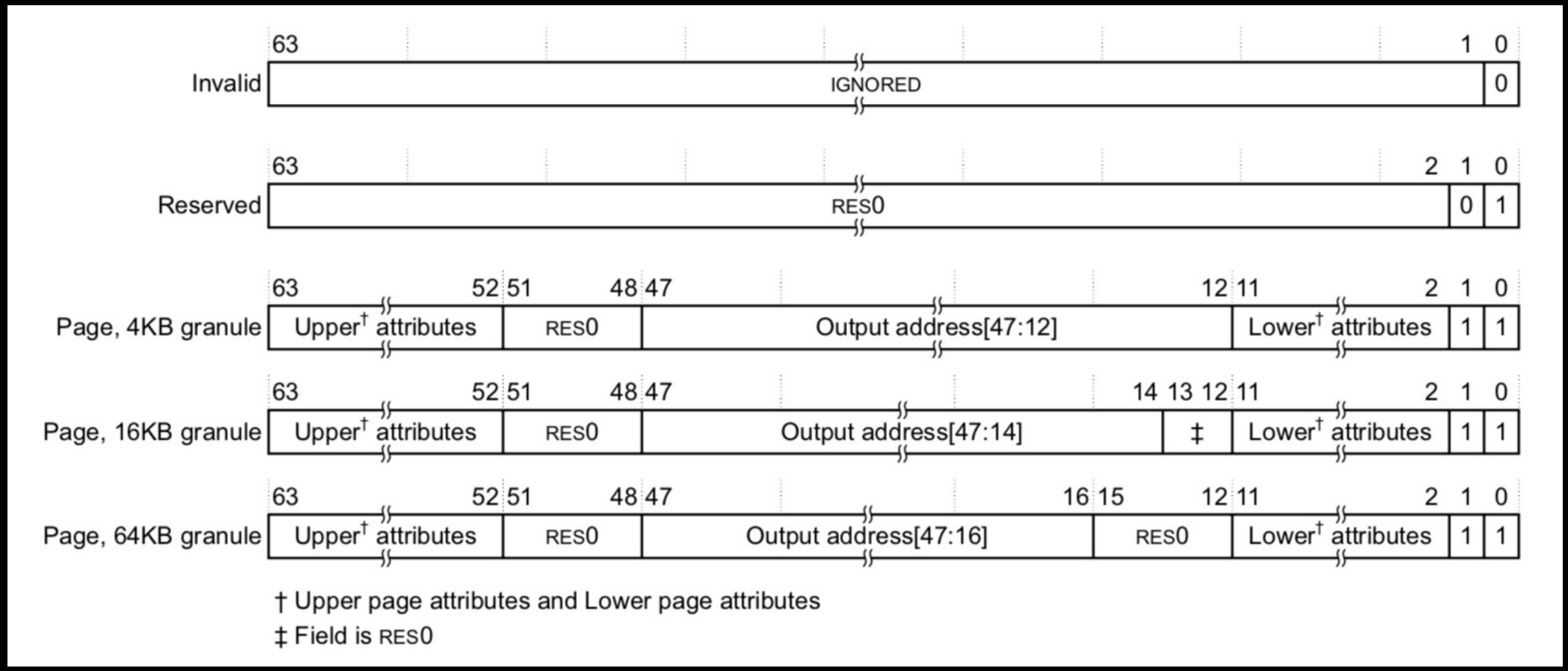
Kernel Space Mirroring Attack

- ARMv8-64 level 0, level 1, and level 2 descriptor formats

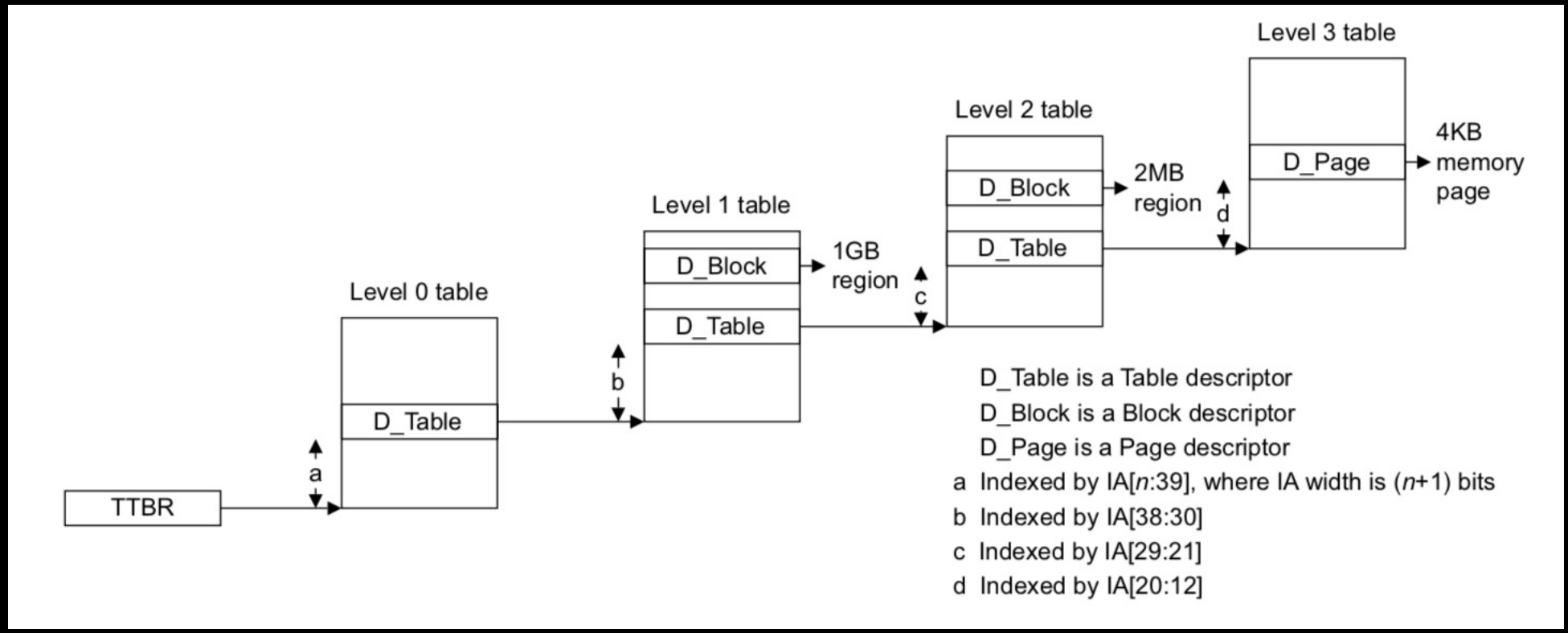


Kernel Space Mirroring Attack

- ARMv8-64 level 3 descriptor format



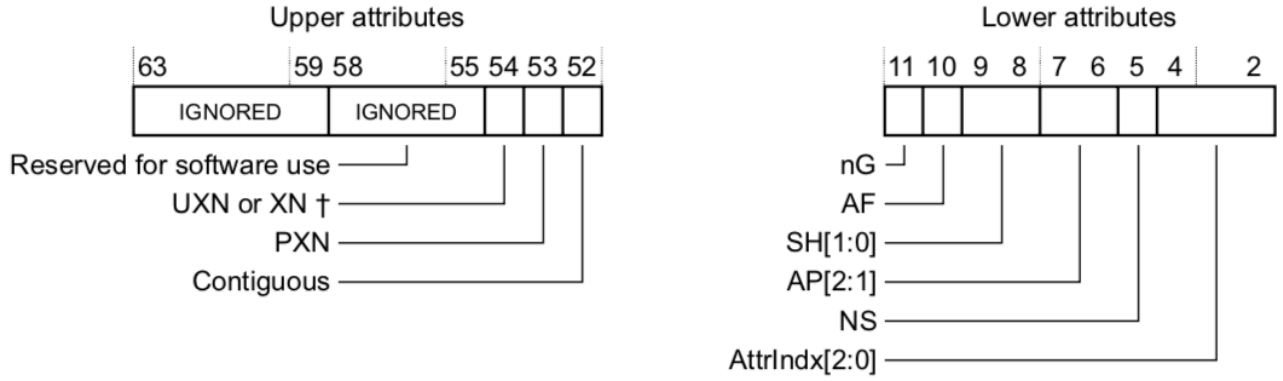
Kernel Space Mirroring Attack



No level 0 table for Android

Kernel Space Mirroring Attack

Attribute fields for VMSAv8-64 stage 1 Block and Page descriptors



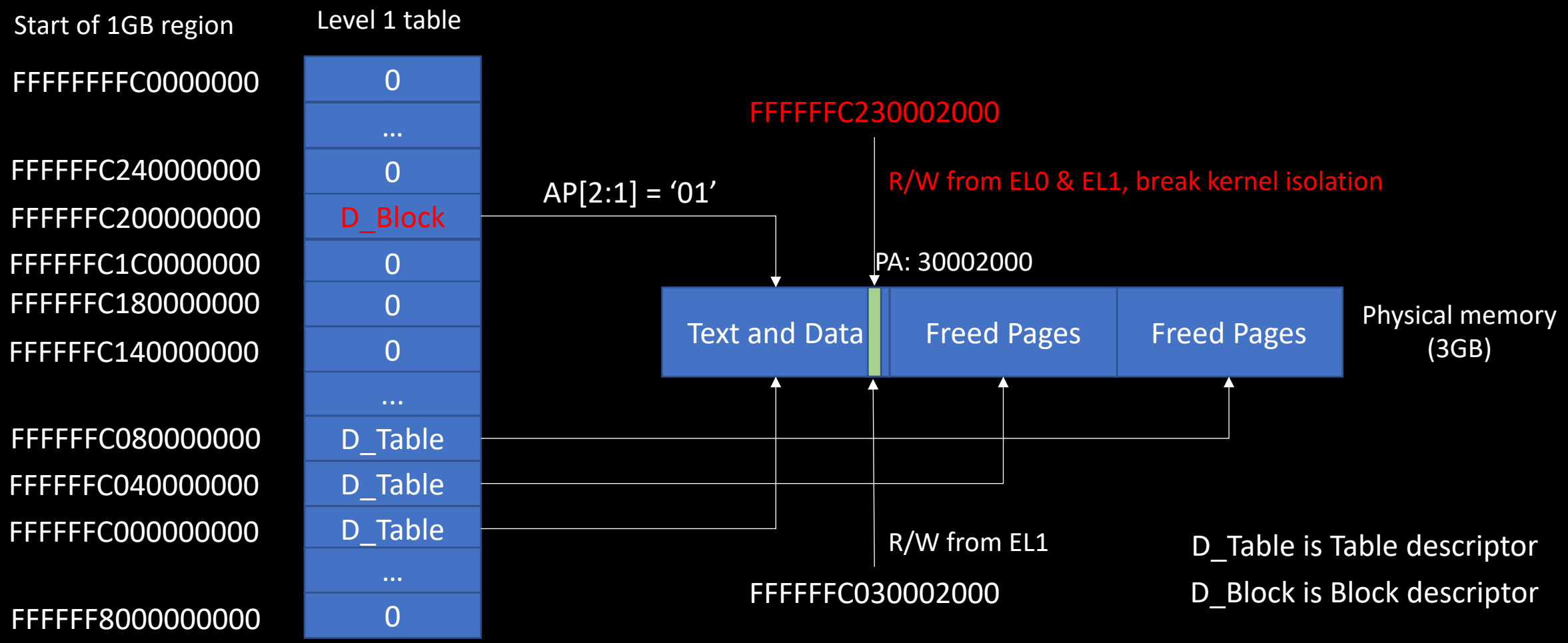
† UXN for the EL1&0 translation regime, XN for the other regimes.

Data access permissions for stage 1 of the EL1&0 translation regime

AP[2:1]	Access from EL1	Access from EL0
00	Read/write	None
01	Read/write	Read/write
10	Read-only	None
11	Read-only	Read-only

- Directly read/write the kernel virtual address

Kernel Space Mirroring Attack



Kernel Space Mirroring Attack

- Where to add a special block descriptor
  - swapper_pg_dir is the pgd for the kernel

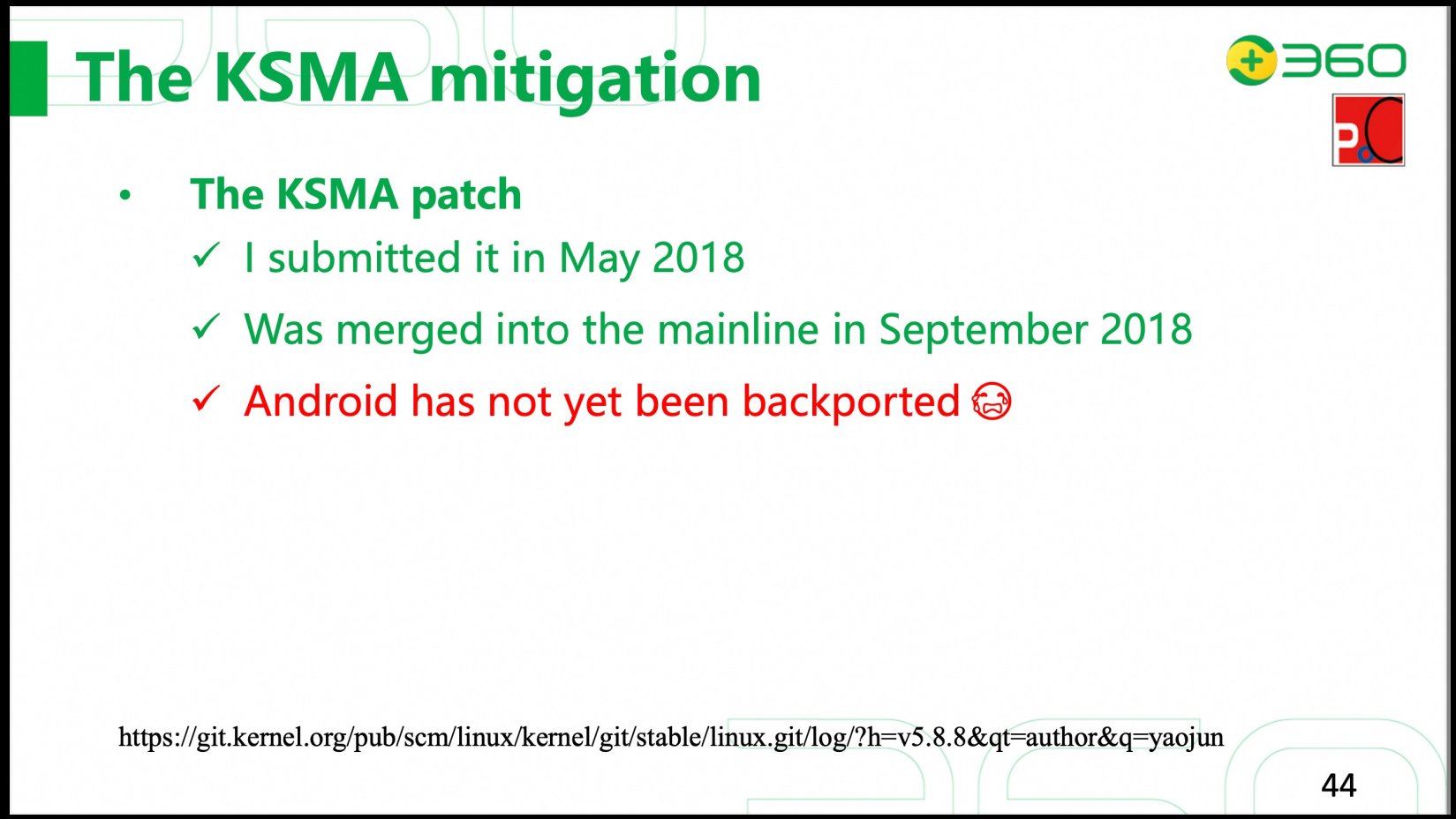
- Kernel mirroring base
  - Entry address
    - (swapper_pg_dir + kernel_slide) + (Kernel_Mirroring_Base / 1GB) * 8

- Kaddr to Mirroring Kaddr
  - Mirroring_kaddr = Kernel_Mirroring_Base + (kaddr - PAGE_OFFSET)

Kernel Space Mirroring Attack

- CVE-2017-0583 / CVE-2017-7533
  - https://i.blackhat.com/briefings/asia/2018/asia-18-WANG-KSMA-Breaking-Android-kernel-isolation-and-Rooting-with-ARM-MMU-features.pdf
- CVE-2020-3680
  - https://github.com/2freeman/Slides/blob/main/PoC-2020-Three%20Dark%20clouds%20over%20the%20Android%20kernel.pdf
- CVE-2020-0423
  - https://www.longterm.io/cve-2020-0423.html
  - https://i.blackhat.com/USA21/Wednesday-Handouts/us-21-Typhoon-Mangkhut-One-Click-Remote-Universal-Root-Formed-With-Two-Vulnerabilities.pdf
- CVE-2021-0399
  - https://conference.hitb.org/hitbsecconf2021sin/materials/D1T1%20-%20%20The%20Art%20of%20Exploiting%20UAF%20by%20Ret2bpf%20in%20Android%20Kernel%20-%20Xingyu%20Jin%20&%20Richard%20Neal.pdf

KSMA defence



https://github.com/2freeman/Slides/blob/main/PoC-2020-Three%20Dark%20clouds%20over%20the%20Android%20kernel.pdf

Agenda

- Introduction
- *GPU version of the KSMA exploitation technique*
- Case study
- Conclusion

GPU Memory Management

Memory Management on Embedded Graphics Processors

Sean Ellis September 11, 2013

6 minute read time.

Our latest world-class embedded graphics processor, the ARM® Mali™-T604 GPU, has excellent memory bandwidth, pixel fill rates to make the mind boggle, and gigaflops of programmable shading power to spare.

We need to keep this engine fuelled with data, and since most of its data comes from memory, we have spent a lot of time and effort designing its Memory Management Unit (MMU). I'd like to show you around its headline features, and explain why a properly designed MMU is so important.

https://community.arm.com/arm-community-blogs/b/graphics-gaming-and-vr-blog/posts/memory-management-on-embedded-graphics-processors

GPU Memory Management

- Features
  - Unified Memory Sharing
    - Same virtual address on both the CPU and GPU
  - Independent address spaces
  - ...

GPU Memory Management

- Features
  - Unified Memory Sharing
    - Same virtual address on both the CPU and GPU
  - Independent address spaces
  - …

- Virtual Memory Management
  - How to manage the virtual addresses?
  - How to manage the physical memory?
  - How the GPU MMU works?

Virtual Address Management

```
struct kbase_va_region {
        struct rb_node rblink;
        struct list_head link;
        u64 start_pfn; // virtual address
        size_t nr_pages;
        size_t initial_commit;
        unsigned long flags; // {KBASE_REG_CPU_WR, KBASE_REG_FREE, …}
        struct kbase_mem_phy_alloc *cpu_alloc;
        struct kbase_mem_phy_alloc *gpu_alloc;
        struct list_head jit_node;
        u16 jit_usage_id;
        u8 jit_bin_id;
        int va_refcnt;
```

Virtual Address Management

```
struct kbase_mem_phy_alloc {
        struct kref         kref;
        atomic_t            gpu_mappings;
        atomic_t            kernel_mappings;
        size_t              nents;
        struct tagged_addr  *pages;
        struct list_head    mappings;
        struct list_head    evict_node;
        size_t              evicted;
        struct kbase_va_region *reg;
        enum kbase_memory_type type;
        struct kbase_vmap_struct *permanent_map;
        u8 properties;
        u8 group_id;
        union {umm, alias, native, user_buf} imported;
```

Virtual Address Management

- struct kbase_reg_zone reg_zone[KBASE_REG_ZONE_MAX];
  - KBASE_REG_ZONE_CUSTOM_VA
  - KBASE_REG_ZONE_SAME_VA
  - KBASE_REG_ZONE_EXEC_VA
  - …

- kbase_context
  - kbase_ctx_reg_zone_init(kctx, KBASE_REG_ZONE_SAME_VA, same_va_base, same_va_pages);
  - kbase_ctx_reg_zone_init(kctx, KBASE_REG_ZONE_EXEC_VA, exec_va_base, KBASE_REG_ZONE_EXEC_VA_SIZE);
  - …

Virtual Address Management

- BASE_MEM_SAME_VA
  - Same virtual address on both the CPU and GPU
  - Force SAME_VA if a 64-bit process
  - Allocate virtual address when mapping the kbase_va_region
  - Map a specific address on the GPU

- Non SAME_VA
  - Allocate virtual address and map it on the GPU immediately
  - Allocate virtual address on the CPU when mapping the kbase_va_region

Physical Page Management

```
struct kbase_mem_pool {
        struct kbase_device *kbdev;
        size_t            cur_size;
        size_t            max_size;
        u8                order;
        u8                group_id;
        spinlock_t        pool_lock;
        struct list_head    page_list;
        struct shrinker     reclaim;

        struct kbase_mem_pool *next_pool;

        bool dying;
        bool dont_reclaim;
};
```

```
struct kbase_mem_pool_group {
        struct kbase_mem_pool small[16];
        struct kbase_mem_pool large[16];
};

int kbase_context_mem_pool_group_init(struct kbase_context
*kctx)
{
        return kbase_mem_pool_group_init(
        &kctx->mem_pools,
        kctx->kbdev,
        &kctx->kbdev->mem_pool_defaults,
        &kctx->kbdev->mem_pools);
}
```

Physical Page Management

- Allocate
  - Step 1: allocate from the kctx->mem_pools. If insufficient, goto step 2
  - Step 2: allocate from the kbdev->mem_pools. If insufficient, goto step 3
  - Step 3: allocate from the kernel
- Free
  - Step 1: add the pages to kctx->mem_pools. If full, goto step 2
  - Step 2: add the pages to kbdev->mem_pools. If full, goto step 3
  - Step 3: free the remaining pages to the kernel

Physical Page Management

- Allocate
  - Step 1: allocate from the kctx->mem_pools. If insufficient, goto step 2
  - Step 2: allocate from the kbdev->mem_pools. If insufficient, goto step 3
  - Step 3: allocate from the kernel
- Free
  - Step 1: add the pages to kctx->mem_pools. If full, goto step 2
  - Step 2: add the pages to kbdev->mem_pools. If full, goto step 3
  - Step 3: free the remaining pages to the kernel
- Shrinker
  - register_shrinker(&kctx->reclaim);
  - register_shrinker(&pool->reclaim);

Memory Management Unit

- No GPU Architecture Reference Manual
  - Learn from the kernel driver
  - Compare with CPU MMU
  - Blind test

Memory Management Unit

- No GPU Architecture Reference Manual
  - Learn from the kernel driver
  - Compare with CPU MMU
  - Blind test
- kctx.mmu

```
struct kbase_mmu_table {
        u64 *mmu_teardown_pages[MIDGARD_MMU_BOTTOMLEVEL];
        struct mutex mmu_lock;
        phys_addr_t pgd;
        u8 group_id;
        struct kbase_context *kctx;
};
```

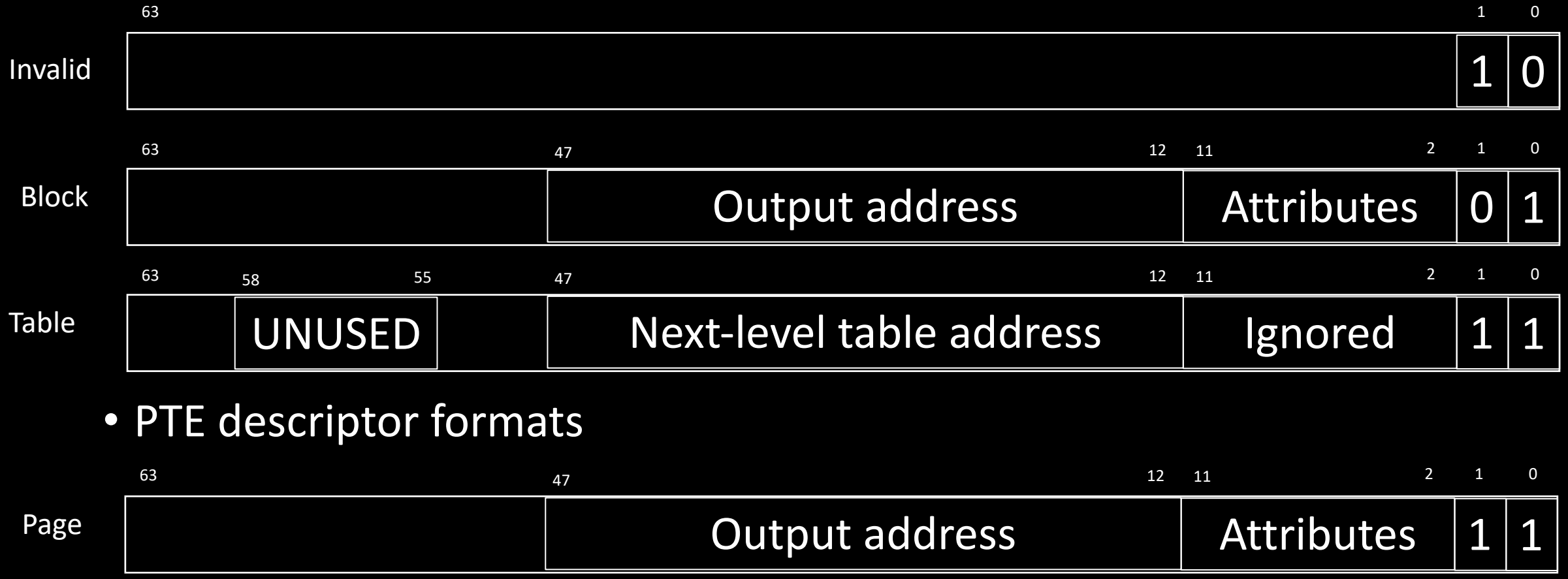
Memory Management Unit

```
#define ENTRY_TYPE_MASK     3ULL
/* For valid ATEs bit 1 = ((level == 3) ? 1 : 0).
 * Valid ATE entries at level 3 are flagged with the value 3.
 * Valid ATE entries at level 0-2 are flagged with the value 1.
 */
#define ENTRY_IS_ATE_L3     3ULL
#define ENTRY_IS_ATE_L02    1ULL
#define ENTRY_IS_INVAL      2ULL
#define ENTRY_IS_PTE        3ULL
```

```
static int ate_is_valid(u64 ate, int const level)
{
    if (level == MIDGARD_MMU_BOTTOMLEVEL)
        return ((ate & ENTRY_TYPE_MASK) == ENTRY_IS_ATE_L3);
    else
        return ((ate & ENTRY_TYPE_MASK) == ENTRY_IS_ATE_L02);
}

static int pte_is_valid(u64 pte, int const level)
{
    /* PTEs cannot exist at the bottom level */
    if (level == MIDGARD_MMU_BOTTOMLEVEL)
        return false;
    return ((pte & ENTRY_TYPE_MASK) == ENTRY_IS_PTE);
}
```

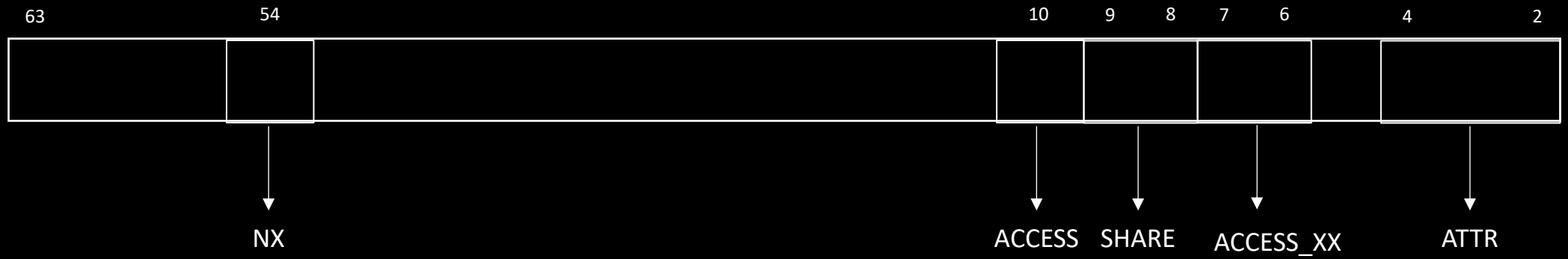
Memory Management Unit

- ATE descriptor formats



- PTE descriptor formats

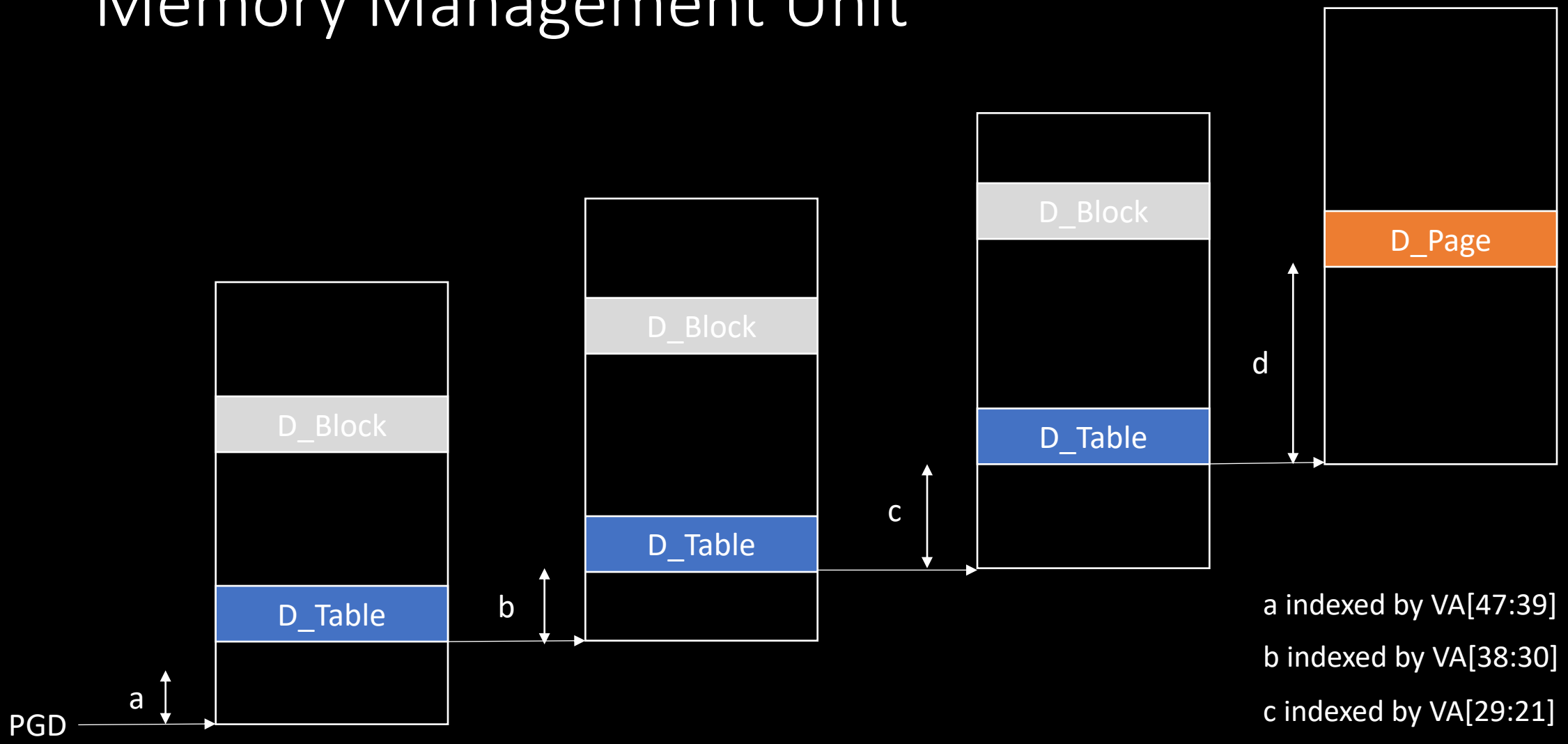
Memory Management Unit



- ATTR(4:2)
  - KBASE_REG_MEMATTR_VALUE(flags) << 2;
- ACCESS_XX(7:6)
  - KBASE_REG_GPU_WR -> ENTRY_ACCESS_RW
  - KBASE_REG_GPU_RD -> ENTRY_ACCESS_RO
- SHARE(9:8)
  - KBASE_REG_SHARE_BOTH -> SHARE_BOTH_BITS
  - KBASE_REG_SHARE_IN -> SHARE_INNER_BITS
- ACCESS(10)
  - ENTRY_ACCESS_BIT
- NX(54)
  - KBASE_REG_GPU_NX -> ENTRY_NX_BIT

```
#define ENTRY_ATTR_BITS (7ULL << 2)
#define ENTRY_ACCESS_RW (1ULL << 6)
#define ENTRY_ACCESS_RO (3ULL << 6)
#define ENTRY_SHARE_BITS (3ULL << 8)
#define ENTRY_ACCESS_BIT (1ULL << 10)
#define ENTRY_NX_BIT (1ULL << 54)
enum kbase_share_attr_bits {
        SHARE_BOTH_BITS = (2ULL << 8),
        SHARE_INNER_BITS = (3ULL << 8)
};
```

Memory Management Unit



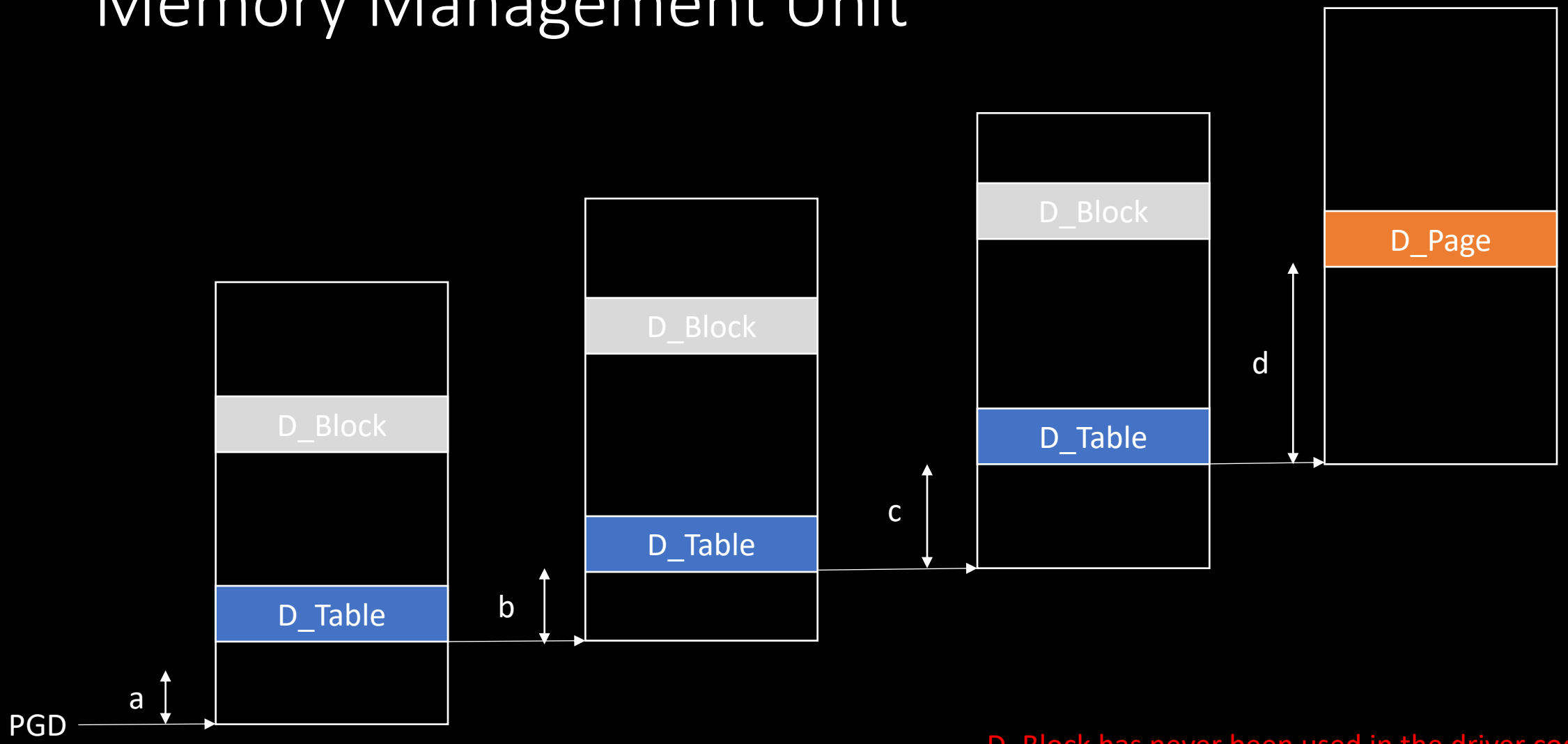
a indexed by VA[47:39]

b indexed by VA[38:30]

c indexed by VA[29:21]

d indexed by VA[20:12]

Memory Management Unit



D_Block has never been used in the driver code

Memory Management Unit

- MMU entries can be dumped
  - Leak the physical page frames(including zero page)
  - Fixed and guarded by non-default config

```
2850        case PFN_DOWN(BASE_MEM_MMU_DUMP_HANDLE):
2851    #if defined(CONFIG_MALI_VECTOR_DUMP)
2852            /* MMU dump */
2853            err = kbase_mmu_dump_mmap(kctx, vma, &reg, &kaddr);
2854            if (err != 0)
2855                goto out_unlock;
2856            /* free the region on munmap */
2857            free_on_close = 1;
2858            break;
2859    #else
```

Memory Management Unit

- Cache maintenance
  - Cache line is made of 64 bytes (8 page table entries)
  - MMU lock region is a self-aligned region whose size is a power of 2

- Example: va=0x4F000 num_pages=2
  - Address range between 0x4F000 and 0x50FFF
  - 0x4F000 falls into the [0x48000, 0x4FFFF]
  - 0x50000 falls into the [0x50000, 0x57FFF]
  - [0x40000, 0x5FFFF]

Memory Management Unit

- Cache maintenance
  - Cache line is made of 64 bytes (8 page table entries)
  - MMU lock region is a self-aligned region whose size is a power of 2
  - Align to 2MB boundaries

```
if (!kbase_ctx_flag(kctx, KCTX_COMPAT)) {
    high_limit =
        min_t(unsigned long, mm->mmap_base, same_va_end_addr);

    /* If there's enough (> 33 bits) of GPU VA space, align
     * to 2MB boundaries.
     */
    if (kctx->kbdev->gpu_props.mmu.va_bits > 33) {
        if (len >= SZ_2M) {
            align_offset = SZ_2M;
            align_mask = SZ_2M - 1;
        }
    }
```

Block descriptor

- One of the key steps of KSMA is crafting a valid block entry
  - Figure out whether the GPU MMU supports block descriptor or not
  - Which level? (L2-2MB, L1-1GB, L0-512GB)
- kbase_mmu_dump_mmap

```
PGD(level: 0): 0(next PGD: 8da80a000)
 PGD(level: 1): 7f00000000(next PGD: 8da686000)
  PGD(level: 2): 7f00000000(next PGD: 8da685000)
    Range: 7f00000000-7f00020000: _*******************************
    Range: 7f00020000-7f00040000: **********************__________
    Range: 7f00040000-7f00060000: ____________________________________
    Range: 7f00060000-7f00080000: ____________________________________
    Range: 7f00080000-7f000a0000: ____________________________________
    Range: 7f000a0000-7f000c0000: ____________________________________
    Range: 7f000c0000-7f000e0000: ____________________________________
    Range: 7f000e0000-7f00100000: ____________________________________
    Range: 7f00100000-7f00120000: ____________________________________
    Range: 7f00120000-7f00140000: ____________________________________
    Range: 7f00140000-7f00160000: ____________________________________
    Range: 7f00160000-7f00180000: ____________________________________
    Range: 7f00180000-7f001a0000: ____________________________________
    Range: 7f001a0000-7f001c0000: ____________________________________
    Range: 7f001c0000-7f001e0000: ____________________________________
    Range: 7f001e0000-7f00200000: ____________________________________
```

“-” denotes Invalid descriptor

“*” denotes Page descriptor

Block descriptor

- Add a block descriptor to L2
  - ENTRY_NX_BIT | (PA_aligned & 0xfffffffff000LL) |ENTRY_ACCESS_BIT | ENTRY_ACCESS_RW | 0x01

Block descriptor

- Add a block descriptor to L2
  - ENTRY_NX_BIT | (PA_aligned & 0xfffffffff000LL) |ENTRY_ACCESS_BIT | ENTRY_ACCESS_RW | 0x01
  - Leak the PGD of L2
  - Write this descriptor to the associated physical page

```
PGD(level: 0): 0(next PGD: 8da80a000)
 PGD(level: 1): 5ec0000000(next PGD: 8da809000)
  PGD(level: 2) BLOCK: 5ef4000000 - 5ef4200000(40000080000441)
  PGD(level: 2): 5ef7a00000(next PGD: acd23000)
     Range: 5ef7a00000-5ef7c00000 is full
 PGD(level: 1): 7f00000000(next PGD: 8da686000)
  PGD(level: 2): 7f00000000(next PGD: 8da685000)
     Range: 7f00000000-7f00020000: _*******************************
     Range: 7f00020000-7f00040000: ***********************_________
     Range: 7f00040000-7f00060000: __________________________________
     Range: 7f00060000-7f00080000: __________________________________
     Range: 7f00080000-7f000a0000: __________________________________
     Range: 7f000a0000-7f000c0000: __________________________________
```

```
80000000-901fffff : System RAM
  80000000-82c5ffff : Kernel code
  82c60000-82ecffff : reserved
  82ed0000-831affff : Kernel data
  85fff000-85ffffff : reserved
  86000000-880a6fff : reserved
  8a000000-8a053fff : reserved
  90000000-90039fff : reserved
```

Block descriptor

- No kbase_va_region and kbase_mem_phy_alloc is associated with this descriptor
  - determined by the GPU whether this descriptor is valid or not

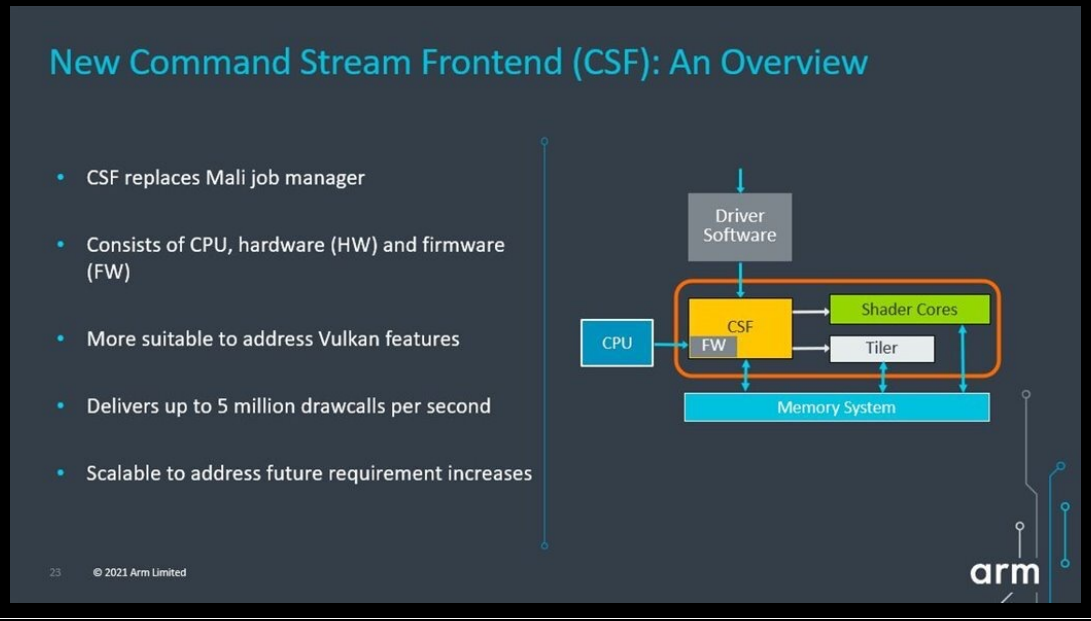
```
PGD(level: 0): 0(next PGD: 8da80a000)
 PGD(level: 1): 5ec0000000(next PGD: 8da809000)
  PGD(level: 2) BLOCK: 5ef4000000 - 5ef4200000(40000080000441)
  PGD(level: 2): 5ef7a00000(next PGD: acd23000)
     Range: 5ef7a00000-5ef7c00000 is full
 PGD(level: 1): 7f00000000(next PGD: 8da686000)
  PGD(level: 2): 7f00000000(next PGD: 8da685000)
     Range: 7f00000000-7f00020000: _*******************************
     Range: 7f00020000-7f00040000: ***********************_________
     Range: 7f00040000-7f00060000: ___________________________________
     Range: 7f00060000-7f00080000: ___________________________________
     Range: 7f00080000-7f000a0000: ___________________________________
     Range: 7f000a0000-7f000c0000: ___________________________________
```

GPU READ/WRITE

- Require reverse-engineering the GPU instruction sets
  - https://gitlab.freedesktop.org/panfrost
  - https://github.blog/2022-07-27-corrupting-memory-without-memory-corruption/

GPU READ/WRITE

- Require reverse-engineering the GPU instruction sets
  - https://gitlab.freedesktop.org/panfrost
  - https://github.blog/2022-07-27-corrupting-memory-without-memory-corruption/
- New features and enhancements




```
#if MALI_USE_CSF
    case KBASE_IOCTL_CS_EVENT_SIGNAL:
        KBASE_HANDLE_IOCTL(KBASE_IOCTL_CS_EVENT_SIGNAL,
                kbasep_cs_event_signal,
                kctx);
        break;
    case KBASE_IOCTL_CS_QUEUE_REGISTER:
        KBASE_HANDLE_IOCTL_IN(KBASE_IOCTL_CS_QUEUE_REGISTER,
                kbasep_cs_queue_register,
                struct kbase_ioctl_cs_queue_register,
                kctx);
        break;
    case KBASE_IOCTL_CS_QUEUE_REGISTER_EX:
        KBASE_HANDLE_IOCTL_IN(KBASE_IOCTL_CS_QUEUE_REGISTER_EX,
                kbasep_cs_queue_register_ex,
                struct kbase_ioctl_cs_queue_register_ex,
                kctx);
        break;
```

GPU READ/WRITE

- OpenCL(Open Computing Language)
  - A framework for writing programs that execute across heterogeneous platforms consisting of central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other processors or hardware accelerators
  - Provides a standard interface for parallel computing using task- and data-based parallelism
  - Specifies programming languages (based on C99, C++14 and C++17) for programming abovementioned devices

GPU READ/WRITE

```
const char* gpu_code =
    "__kernel void rw_mem(__global unsigned long *p0, __global unsigned long *p1, __global unsigned long *p2) {”  // p0 – dest, p1 – src, p2 – rw_flag
    "   size_t idx = get_global_id(0);"
    "   if (p2[idx]) {” // write
    "       __global unsigned long *addr = (__global unsigned long)(p0[idx]);"
    "       addr[0] = p1[idx];"
    "   } else {” // read
    "       __global unsigned long *addr = (__global unsigned long *)(p1[idx]);"
    "       p0[idx] = addr[0];"
    "   }"
    "}";
```

GPU READ/WRITE

- How to use the OpenCL
  - Find all the needed functions(clCreateBuffer/clSetKernelArg,etc.) from libOpenCL.so
  - Create the GPU buffer for P0-P2 (clCreateBuffer)
  - Set the parameters of kernel function - rw_mem (clSetKernelArg)
  - Kick off the GPU work (clEnqueueNDRangeKernel)
  - Read the result (clEnqueueReadBuffer)

- For a valid kbase_va_region, it works well

GPU READ/WRITE

- clEnqueueReadBuffer always fails

```
[ 9589.105779] mali 1c500000.mali: Unhandled Page fault in AS4 at VA 0x0000005EF4200000
    Reason: kbase_mmu_page_fault_worker: Memory is not mapped on the GPU
    raw fault status: 0x10002C2
    exception type 0xC2: TRANSLATION_FAULT
    access type 0x2: READ
    source id 0x100
    pid: 18488
[ 9589.105843] mali 1c500000.mali: error detected from slot 1, job status 0x00000004 (TERMINATED)
[ 9589.106014] mali 1c500000.mali: t6xx: GPU fault 0x04 from job slot 1
```

```
kbase_gpu_vm_lock(kctx);

region = kbase_region_tracker_find_region_enclosing_address(kctx,
        fault->addr);
if (kbase_is_region_invalid_or_free(region)) {
    kbase_gpu_vm_unlock(kctx);
    kbase_mmu_report_fault_and_kill(kctx, faulting_as,
            "Memory is not mapped on the GPU", fault);
    goto fault_done;
}
```

No kbase_va_region and kbase_mem_phy_alloc is associated with the block descriptor.

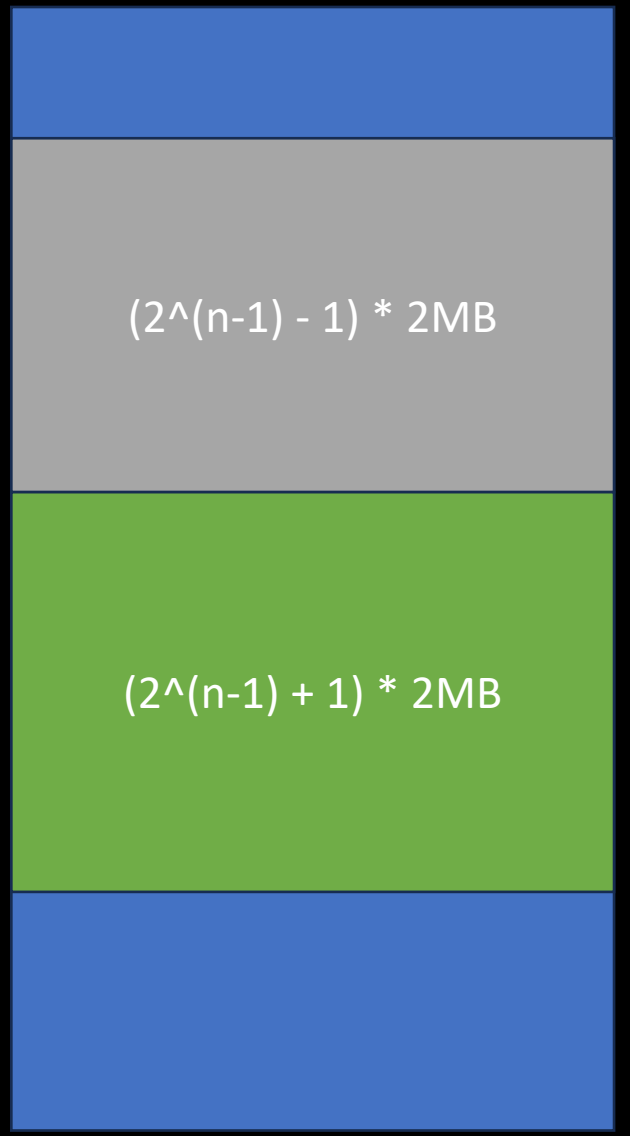
GPU READ/WRITE

- MMU cache flush
  - Insert/teardown pages can flush the MMU cache
  - The block descriptor is invisible from CPU side

GPU READ/WRITE

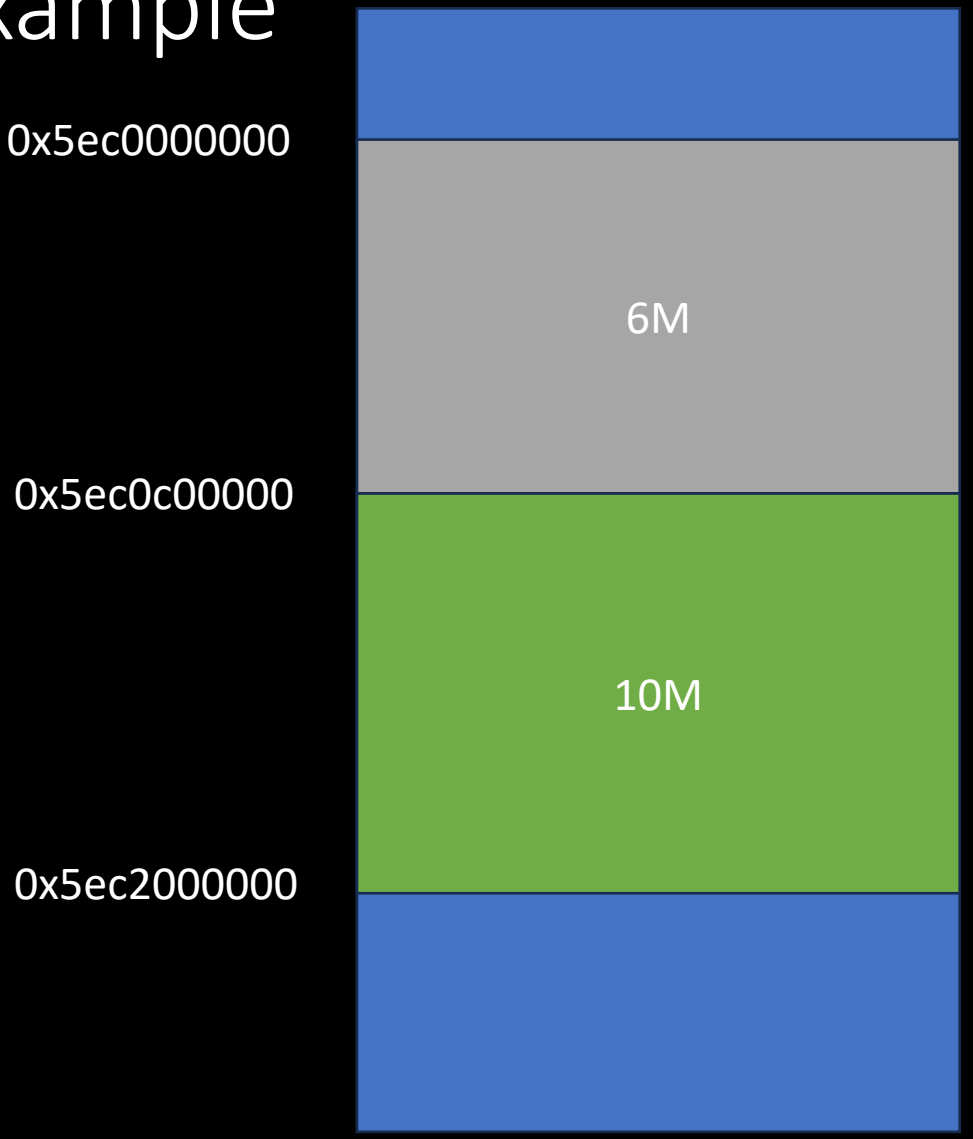
- MMU cache flush
  - Insert/teardown pages can flush the MMU cache
  - The block descriptor is invisible from CPU side

- Side-channel cache flush
  - Valid GPU VA start: (2^(n-1) + 1) * 2MB
  - Fake GPU VA start: (2^(n-1) - 1) * 2MB
  - MMU lock region: (2^n) * 2MB



Side-channel cache flush example

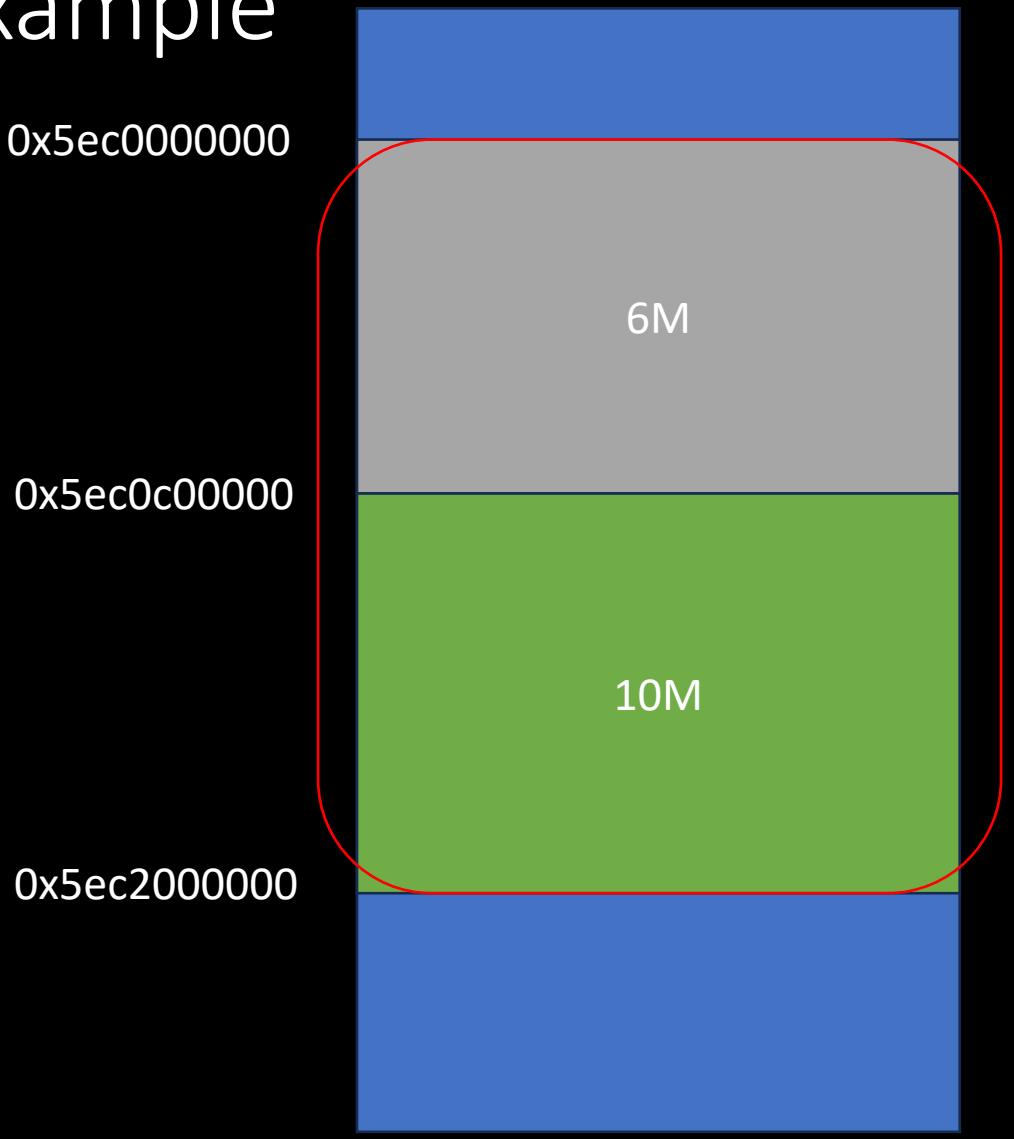
- Valid GPU VA
  - [0x5ec0c00000, 0x5ec2000000)
- Fake GPU VA
  - [0x5ec0000000, 0x5ec0c00000)



Side-channel cache flush example

- Valid GPU VA
  - [0x5ec0c00000, 0x5ec2000000)
- Fake GPU VA
  - [0x5ec0000000, 0x5ec0c00000)

- Commit or shrink the valid VA
  - Lock region [0x5ec0000000, 0x5ec2000000)



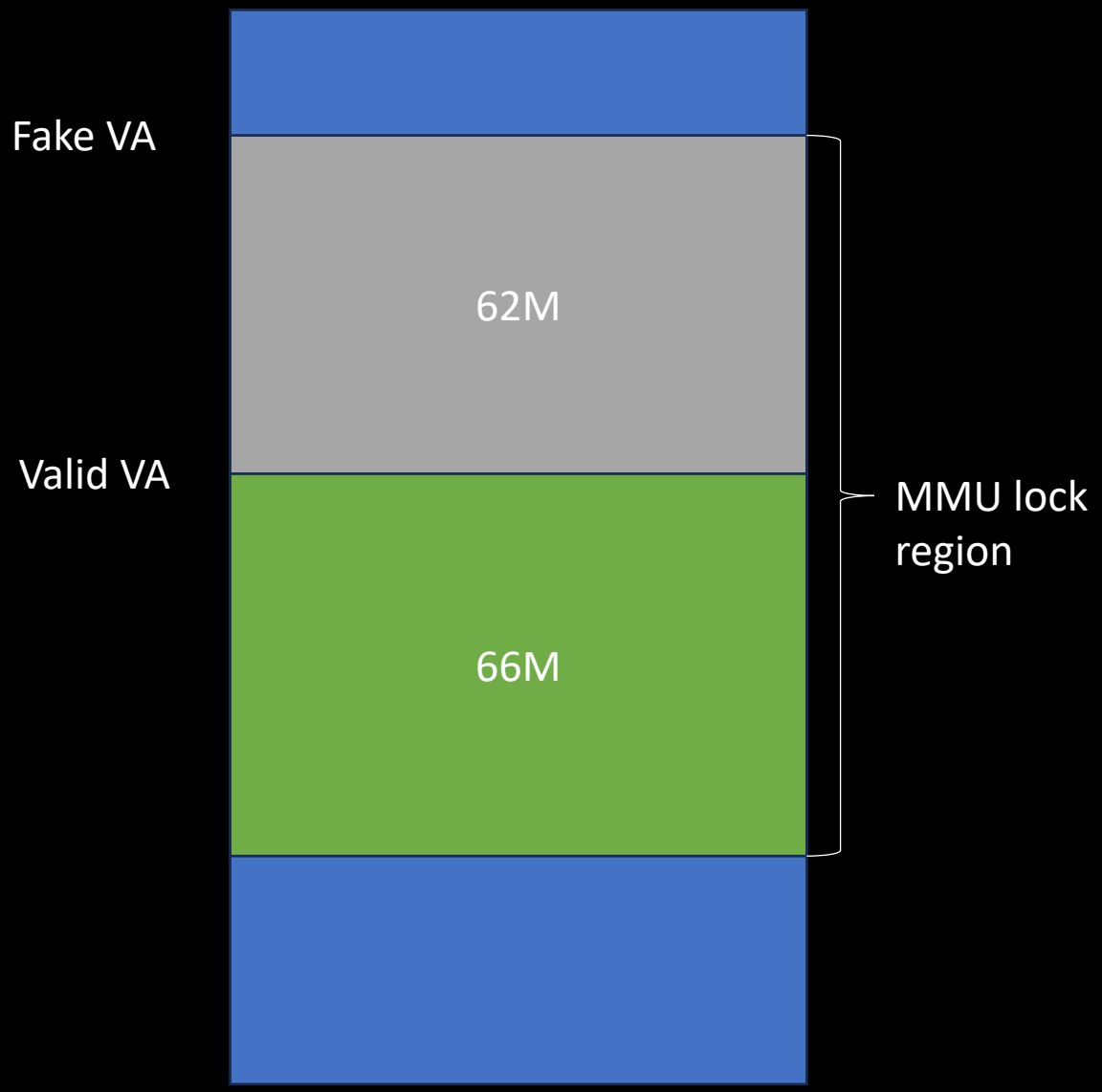
GPU READ/WRITE

- Is it possible to add the L1 or L0 block descriptor?
  - Absolutely yes

- Problems
  - Shaping the GPU VA layout may be more complex
  - Using too many physical pages can trigger the Out of Memory

GPU version of KSMA

- Shape the GPU VA layout
  - Spray the GPU VAs without pages
- Full physical memory access
  - 1. Add 31 block descriptors
  - 2. Commit/shrink the valid VA
  - 3. Read/write the fake VA
  - 4. GOTO Step 1

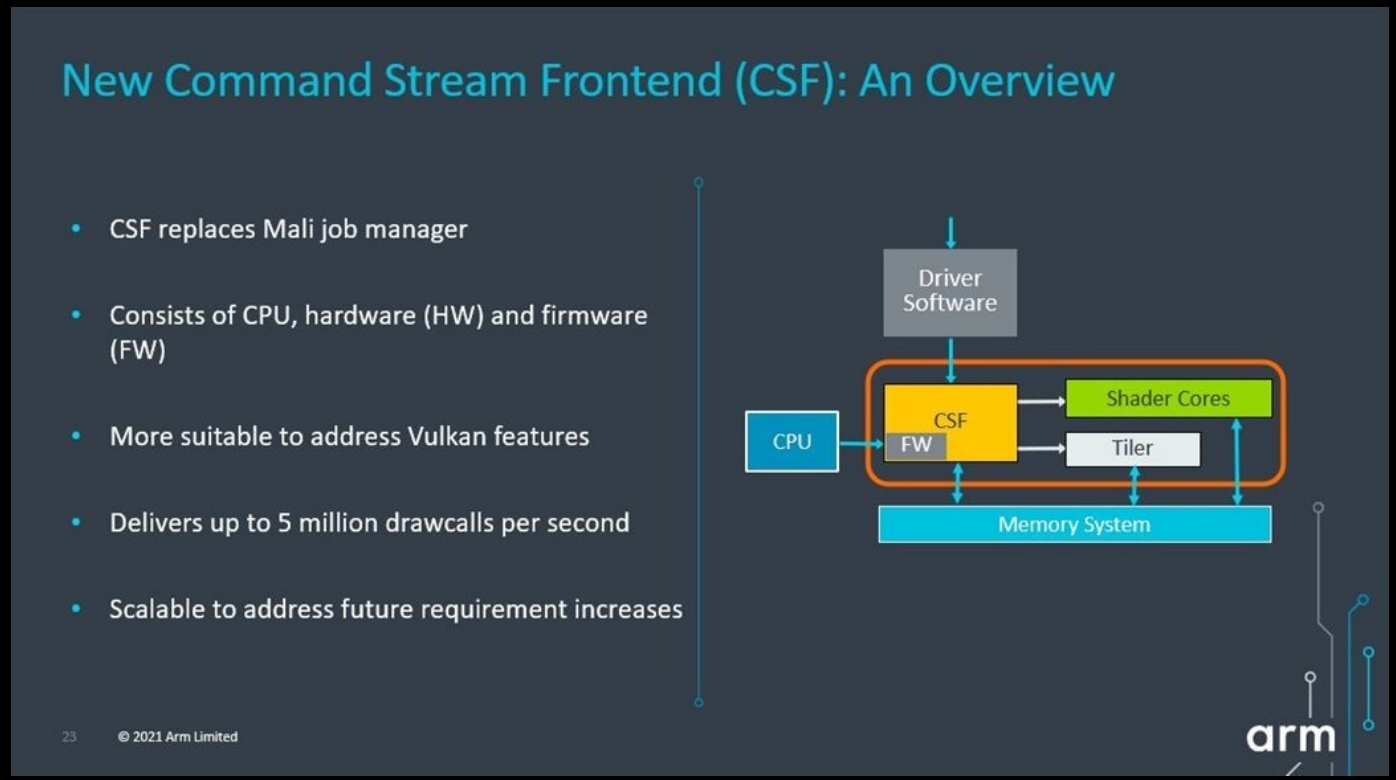


Agenda

- Introduction
- GPU version of the KSMA exploitation technique
- *Case study*
- Conclusion

Vulnerability analysis

- New features and enhancements
  - New code
  - MTK dimensity 9000 / Google tensor GS201(Pixel 7)



Vulnerability analysis

- KBASE_IOCTL_CS_QUEUE_REGISTER

```
queue_addr = reg->buffer_gpu_addr;
queue_size = reg->buffer_size >> PAGE_SHIFT;
/* Check if queue is already registered */
if (find_queue(kctx, queue_addr) != NULL) {
        ret = -EINVAL;
        goto out;
}
region = kbase_region_tracker_find_region_enclosing_address(kctx,    queue_addr);

region->flags |= KBASE_REG_NO_USER_FREE;
```

Vulnerability analysis

- KBASE_IOCTL_CS_QUEUE_TERMINATE

```
queue = find_queue(kctx, term->buffer_gpu_addr);
if (queue) {
        unbind_queue(kctx, queue);
        if (!WARN_ON(!queue->queue_reg)) {
                /* After this the Userspace would be able to free the
                 * memory for GPU queue. In case the Userspace missed
                 * terminating the queue, the cleanup will happen on
                 * context termination where tear down of region tracker
                 * would free up the GPU queue memory.
                 */
                queue->queue_reg->flags &= ~KBASE_REG_NO_USER_FREE;
```

Vulnerability analysis

```
int kbase_mem_free_region(struct kbase_context *kctx, struct kbase_va_region *reg)
{
        int err;

        KBASE_DEBUG_ASSERT(kctx != NULL);
        KBASE_DEBUG_ASSERT(reg != NULL);
        dev_dbg(kctx->kbdev->dev, "%s %pK in kctx %pK\n",
                __func__, (void *)reg, (void *)kctx);
        lockdep_assert_held(&kctx->reg_lock);

        if (reg->flags & KBASE_REG_NO_USER_FREE) {
                dev_warn(kctx->kbdev->dev, "Attempt to free GPU memory whose freeing by user space is
forbidden!\n");
                return -EINVAL;
        }
```

Vulnerability analysis

- Unconditionally clear the KBASE_REG_NO_USER_FREE flag
  - KBASE_IOCTL_CS_QUEUE_REGISTER
  - KBASE_IOCTL_CS_QUEUE_TERMINATE

Vulnerability analysis

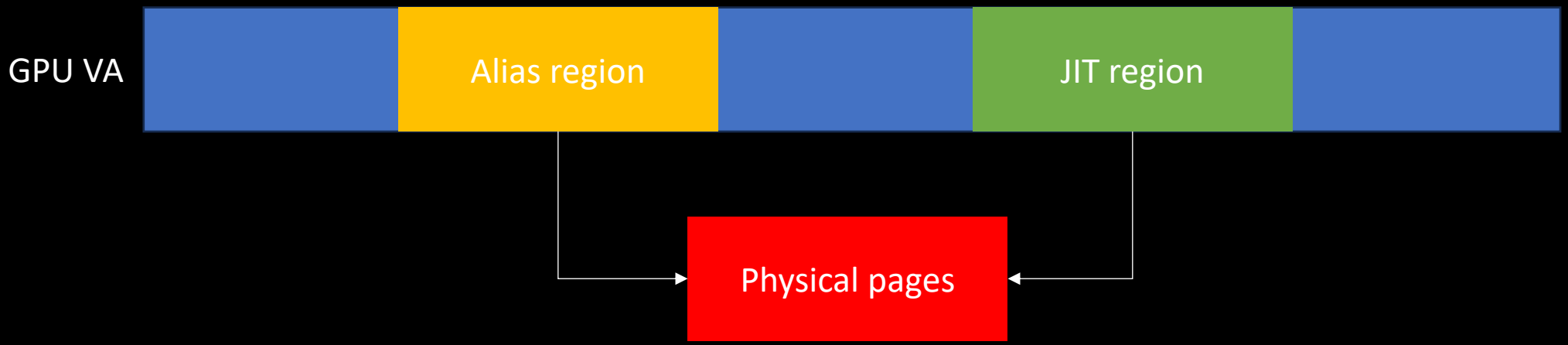
- Unconditionally clear the KBASE_REG_NO_USER_FREE flag
  - KBASE_IOCTL_CS_QUEUE_REGISTER
  - KBASE_IOCTL_CS_QUEUE_TERMINATE
- Impact
  - A region with KBASE_REG_NO_USER_FREE can be freed by user space
  - A region with KBASE_REG_NO_USER_FREE can be aliased(KBASE_IOCTL_MEM_ALIAS)

```
if (aliasing_reg->flags & KBASE_REG_NO_USER_FREE)
    goto bad_handle; /* JIT regions can't be
                      * aliased. NO_USER_FREE flag
                      * covers the entire lifetime
                      * of JIT regions. The other
                      * types of regions covered
                      * by this flag also shall
                      * not be aliased.
                      */
```

Vulnerability analysis

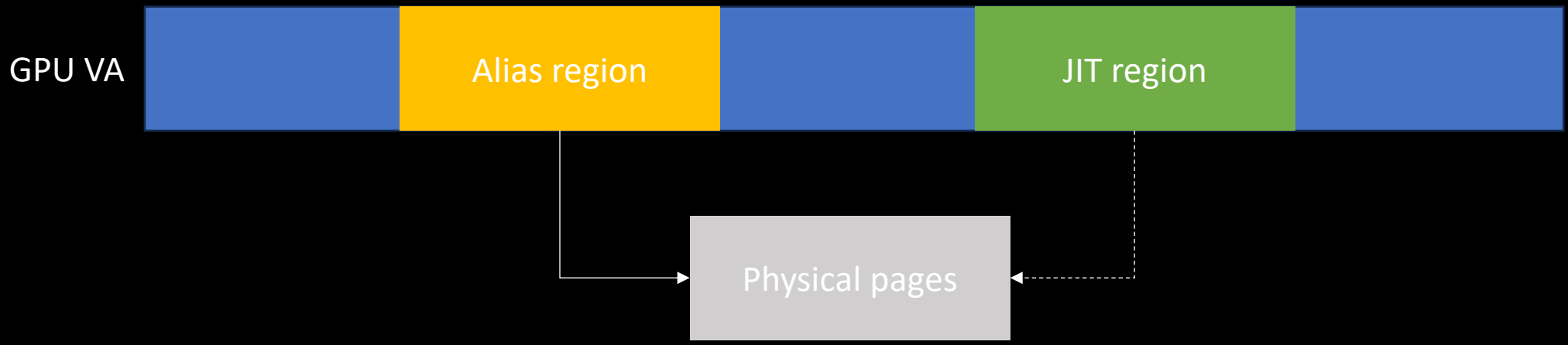
- JIT region can be aliased

  BASE_MEM_PROT_CPU_RD | BASE_MEM_PROT_GPU_RD |
  BASE_MEM_PROT_GPU_WR | BASE_MEM_GROW_ON_GPF |
  BASE_MEM_COHERENT_LOCAL | BASEP_MEM_NO_USER_FREE;



Vulnerability analysis

- JIT region can be freed via BASE_KCPU_COMMAND_TYPE_JIT_FREE
  - The associated physical pages can be reclaimed



Vulnerability analysis

```
void kbase_jit_free(struct kbase_context *kctx, struct kbase_va_region *reg)
{
        old_pages = kbase_reg_current_backed_size(reg);
        if (reg->initial_commit < old_pages) {
                u64 new_size = MAX(reg->initial_commit,
                                div_u64(old_pages * (100 - kctx->trim_level), 100));
                u64 delta = old_pages - new_size;
                if (delta) {
                        mutex_lock(&kctx->reg_lock);
                        kbase_mem_shrink(kctx, reg, old_pages - delta);
                        mutex_unlock(&kctx->reg_lock);
        }

        kbase_mem_shrink_cpu_mapping(kctx, reg, 0, reg->gpu_alloc->nents);
        list_add(&reg->gpu_alloc->evict_node, &kctx->evict_list);
        list_move(&reg->jit_node, &kctx->jit_pool_head);
```

- Tear down the entries of CPU MMU
- Insert the region into the evict_list
- Partially shrink the memory when initial_commit is less than old_pages

Vulnerability analysis

```
// register_shrinker(&kctx->reclaim);

unsigned long kbase_mem_evictable_reclaim_scan_objects(struct
shrinker *s, struct shrink_control *sc)

        // …

         list_for_each_entry_safe(alloc, tmp, &kctx->evict_list,
evict_node) {

                err = kbase_mem_shrink_gpu_mapping(kctx, alloc-
>reg, 0, alloc->nents);

                kbase_free_phy_pages_helper(alloc, alloc->evicted);

        // ...
```

- Tear down the entries of GPU MMU
- Free the physical pages

Vulnerability analysis

```
// register_shrinker(&kctx->reclaim);
unsigned long kbase_mem_evictable_reclaim_scan_objects(struct
shrinker *s, struct shrink_control *sc)
        // …
         list_for_each_entry_safe(alloc, tmp, &kctx->evict_list,
evict_node) {
                err = kbase_mem_shrink_gpu_mapping(kctx, alloc-
>reg, 0, alloc->nents);
                kbase_free_phy_pages_helper(alloc, alloc->evicted);
        // ...
```

- Tear down the entries of GPU MMU
- Free the physical pages
- It requires finding a method to trigger the callback function

Vulnerability analysis

- reg->initial_commit
  - reg->initial_commit = jit_alloc_info.commit_pages
- old_pages = kbase_reg_current_backed_size(reg);
  - reg->cpu_alloc->nents;

Vulnerability analysis

- reg->initial_commit
  - reg->initial_commit = jit_alloc_info.commit_pages
- old_pages = kbase_reg_current_backed_size(reg);
  - reg->cpu_alloc->nents;

- BASE_KCPU_COMMAND_TYPE_JIT_ALLOC
  - jit_alloc_info.va_pages = 1;
  - jit_alloc_info.commit_pages = 0;

  - reg->initial_commit = 0;
  - old_pages = 0;

Vulnerability analysis

- JIT region

  BASE_MEM_PROT_CPU_RD | BASE_MEM_PROT_GPU_RD |
  BASE_MEM_PROT_GPU_WR | BASE_MEM_GROW_ON_GPF |
  BASE_MEM_COHERENT_LOCAL | BASEP_MEM_NO_USER_FREE;

Vulnerability analysis

- JIT region

  BASE_MEM_PROT_CPU_RD | BASE_MEM_PROT_GPU_RD |
  BASE_MEM_PROT_GPU_WR | BASE_MEM_GROW_ON_GPF |
  BASE_MEM_COHERENT_LOCAL | BASEP_MEM_NO_USER_FREE;

- Page fault
  - Read or write the JIT region via OpenCL kernel function
  - reg->initial_commit = 0;
  - old_pages = 1;

Vulnerability analysis

```
void kbase_jit_free(struct kbase_context *kctx, struct kbase_va_region *reg)
{
        old_pages = kbase_reg_current_backed_size(reg);
        if (reg->initial_commit < old_pages) {
                u64 new_size = MAX(reg->initial_commit,
                                div_u64(old_pages * (100 - kctx->trim_level), 100));
                u64 delta = old_pages - new_size;
                if (delta) {
                        mutex_lock(&kctx->reg_lock);
                        kbase_mem_shrink(kctx, reg, old_pages - delta);
                        mutex_unlock(&kctx->reg_lock);
        }

        kbase_mem_shrink_cpu_mapping(kctx, reg, 0, reg->gpu_alloc->nents);
        list_add(&reg->gpu_alloc->evict_node, &kctx->evict_list);
        list_move(&reg->jit_node, &kctx->jit_pool_head);
```

- trim_level is 0
  - new_size = old_pages
  - delta is 0
- trim_level is 100
  - new_size = reg->initial_commit;
  - delta is 1

Vulnerability analysis

```
void kbase_jit_free(struct kbase_context *kctx, struct kbase_va_region *reg)
{
        old_pages = kbase_reg_current_backed_size(reg);
        if (reg->initial_commit < old_pages) {
                u64 new_size = MAX(reg->initial_commit,
                        div_u64(old_pages * (100 - kctx->trim_level), 100));
                u64 delta = old_pages - new_size;
                if (delta) {
                        mutex_lock(&kctx->reg_lock);
                        kbase_mem_shrink(kctx, reg, old_pages - delta);
                        mutex_unlock(&kctx->reg_lock);
        }

        kbase_mem_shrink_cpu_mapping(kctx, reg, 0, reg->gpu_alloc->nents);
        list_add(&reg->gpu_alloc->evict_node, &kctx->evict_list);
        list_move(&reg->jit_node, &kctx->jit_pool_head);
```

- trim_level is 0
  - new_size = old_pages
  - delta is 0
- trim_level is 100
  - new_size = reg->initial_commit;
  - delta is 1
- kctx->trim_level = trim_level;
  - KBASE_IOCTL_MEM_JIT_INIT
  - Always free the physical pages

Fix

```
-       if (kbase_is_region_invalid_or_free(region)) {
+       if (kbase_is_region_invalid_or_free(region) || kbase_is_region_shrinkable(region) ||
+           region->gpu_alloc->type != KBASE_MEM_TYPE_NATIVE) {
                ret = -ENOENT;
                goto out_unlock_vm;
        }
@@ -555,7 +567,7 @@

        queue->kctx = kctx;
        queue->base_addr = queue_addr;
-       queue->queue_reg = region;
+       queue->queue_reg = kbase_va_region_no_user_free_get(kctx, region);
        queue->size = (queue_size << PAGE_SHIFT);
        queue->csi_index = KBASEP_IF_NR_INVALID;
        queue->enabled = false;
@@ -589,7 +601,6 @@

        queue->extract_ofs = 0;

-       region->flags |= KBASE_REG_NO_USER_FREE;
        region->user_data = queue;

        /* Initialize the cs_trace configuration parameters, When buffer_size
@@ -683,16 +694,8 @@
                unbind_queue(kctx, queue);

                kbase_gpu_vm_lock(kctx);
-               if (!WARN_ON(!queue->queue_reg)) {
-                       /* After this the Userspace would be able to free the
-                        * memory for GPU queue. In case the Userspace missed
-                        * terminating the queue, the cleanup will happen on
-                        * context termination where tear down of region tracker
-                        * would free up the GPU queue memory.
-                        */
-                       queue->queue_reg->flags &= ~KBASE_REG_NO_USER_FREE;
+               if (!WARN_ON(!queue->queue_reg))
                        queue->queue_reg->user_data = NULL;
-               }
                kbase_gpu_vm_unlock(kctx);
```

- https://android.googlesource.com/kernel/google-modules/gpu/+/422aa1fad7e63f16000ffb9303e816b54ef3d8ca%5E%21/#F0

Exploit analysis

- Allocate
  - Step 1: allocate from the kctx->mem_pools. If insufficient, goto step 2
  - Step 2: allocate from the kbdev->mem_pools. If insufficient, goto step 3
  - Step 3: allocate from the kernel
- Free
  - Step 1: add the pages to kctx->mem_pools. If full, goto step 2
  - Step 2: add the pages to kbdev->mem_pools. If full, goto step 3
  - Step 3: free the remaining pages to the kernel
- Shrinker
  - register_shrinker(&kctx->reclaim);
  - register_shrinker(&pool->reclaim);

Exploit analysis

- kbase_mmu_insert_pages_no_flush
  - If invalid, allocate one page as the PGD
  - Allocate from kbdev->mem_pools, not from kctx->mem_pools

```
if (!kbdev->mmu_mode->pte_is_valid(page[vpfn], level)) {
    enum kbase_mmu_op_type flush_op = KBASE_MMU_OP_NONE;
    unsigned int current_valid_entries;
    u64 managed_pte;

    target_pgd = kbase_mmu_alloc_pgd(kbdev, mmut);
    if (target_pgd == KBASE_MMU_INVALID_PGD_ADDRESS) {
        dev_dbg(kbdev->dev, "%s: kbase_mmu_alloc_pgd failure\n",
                __func__);
        kunmap(p);
        return -ENOMEM;
    }
```

```
static phys_addr_t kbase_mmu_alloc_pgd(struct kbase_device *kbdev,
        struct kbase_mmu_table *mmut)
{
    u64 *page;
    struct page *p;
    phys_addr_t pgd;

    p = kbase_mem_pool_alloc(&kbdev->mem_pools.small[mmut->group_id]);
    if (!p)
        return KBASE_MMU_INVALID_PGD_ADDRESS;
```

Exploit analysis

- kbase_mmu_insert_pages_no_flush
  - If invalid, allocate one page as the PGD
  - Allocate from kbdev->mem_pools, not from kctx->mem_pools
  - It’s possible to reuse the freed pages as the PGD

```
if (!kbdev->mmu_mode->pte_is_valid(page[vpfn], level)) {
    enum kbase_mmu_op_type flush_op = KBASE_MMU_OP_NONE;
    unsigned int current_valid_entries;
    u64 managed_pte;

    target_pgd = kbase_mmu_alloc_pgd(kbdev, mmut);
    if (target_pgd == KBASE_MMU_INVALID_PGD_ADDRESS) {
        dev_dbg(kbdev->dev, "%s: kbase_mmu_alloc_pgd failure\n",
                __func__);
        kunmap(p);
        return -ENOMEM;
    }
```

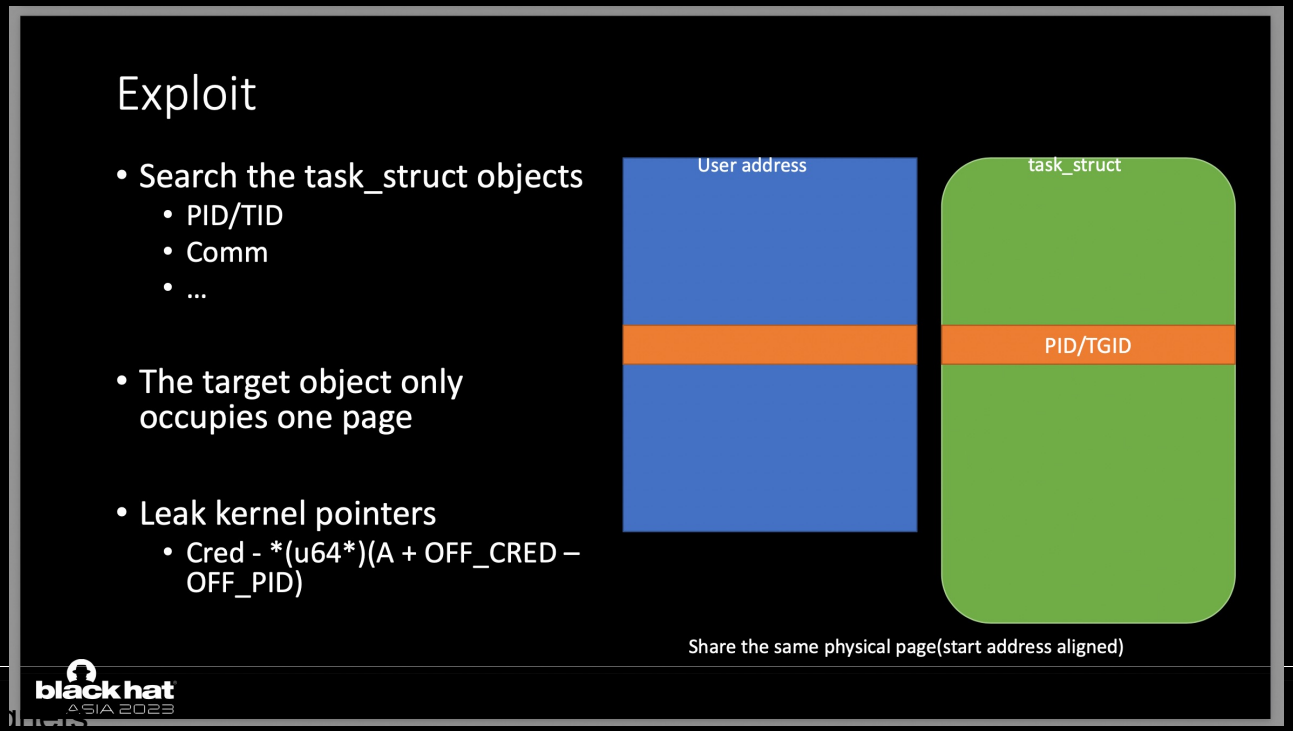
```
static phys_addr_t kbase_mmu_alloc_pgd(struct kbase_device *kbdev,
        struct kbase_mmu_table *mmut)
{
    u64 *page;
    struct page *p;
    phys_addr_t pgd;

    p = kbase_mem_pool_alloc(&kbdev->mem_pools.small[mmut->group_id]);
    if (!p)
        return KBASE_MMU_INVALID_PGD_ADDRESS;
```

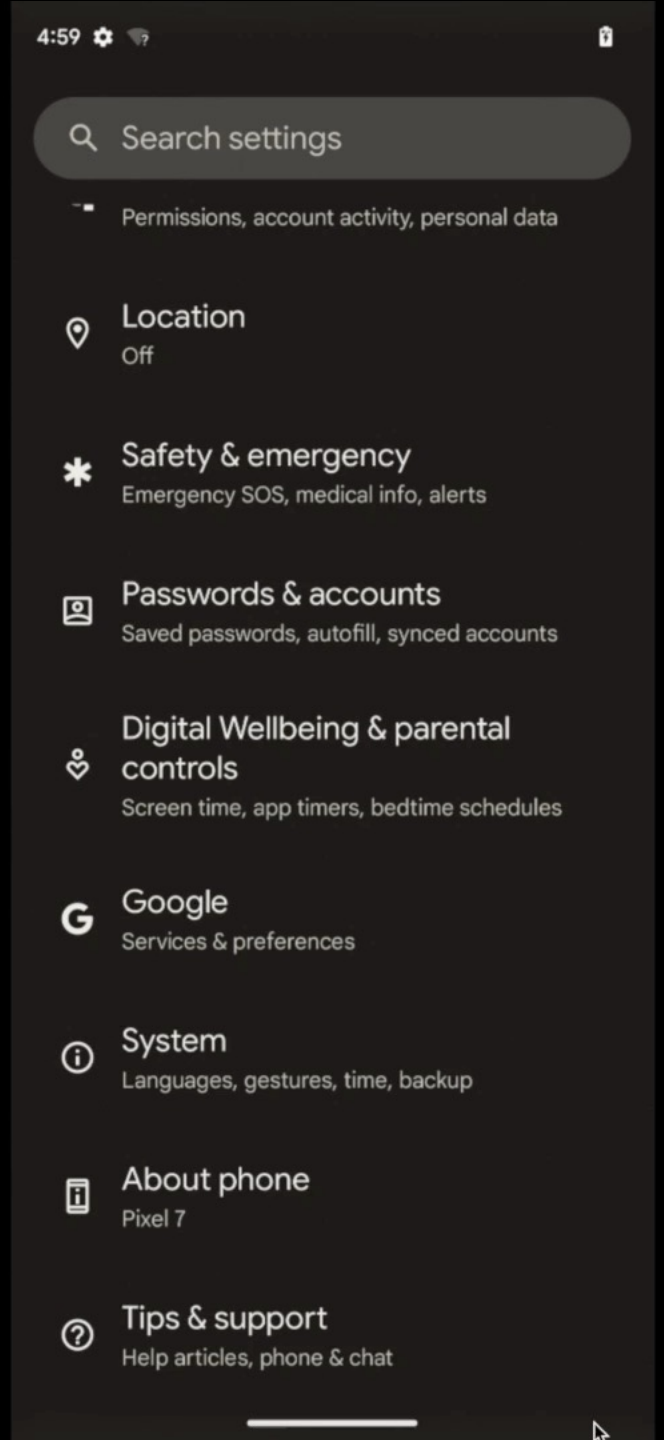
Exploit analysis

- Put it together
  - Initialize the kctx with trim_level = 100
  - Allocate the dummy region with (SZ_64M >> PAGE_SHIFT) pages
    - #define KBASE_MEM_POOL_MAX_SIZE_KCTX (SZ_64M >> PAGE_SHIFT)
  - Allocate a JIT region(VA_SIZE=1, nents=0)
  - Write a value to JIT region and trigger the page fault (VA_SIZE=1, nents=1)
  - Clear the KBASE_REG_NO_USER_FREE flag of the JIT region
  - Create an alias region of the JIT region
  - Free the dummy region
    - kctx->mem_pools is full
  - Shape the VA layout and allocate a candidate region with no page
    - Spray the VA with no page
  - Free the JIT region
    - Pages will be add to kbdev->mem_pools
  - Commit the pages to the candidate region
    - Allocate some pages for PGDs
  - Read or write the specific PGD page via the alias region
    - BASE_MEM_PROT_GPU_RD | BASE_MEM_PROT_GPU_WR | BASE_MEM_PROT_CPU_RD
  - Apply the KSMA exploit technique and access the full memory

Exploit analysis

- Devices without any protection for kernel code/data area
  - Patch the kernel code
- Find a target process and patch the cred pointer
  - GPU accelerate the search!




4:59
Search settings
Permissions, account activity, personal data
Location
Off
Safety & emergency
Emergency SOS, medical info, alerts
Passwords & accounts
Saved passwords, autofill, synced accounts
Digital Wellbeing & parental controls
Screen time, app timers, bedtime schedules
Google
Services & preferences
System
Languages, gestures, time, backup
About phone
Pixel 7
Tips & support
Help articles, phone & chat

Agenda

- Introduction
- GPU version of the KSMA exploitation technique
- Case study
- *Conclusion*

Black Hat sound bytes

- The GPU memory management has been fully discussed.
- The GPU version of the KSMA exploitation technique has been detailed. The bug as case study has been fully discussed.
- With more and more both hardware and software mitigations, Android rooting needs better bug and more advanced exploitation technique.

References

[1] https://i.blackhat.com/briefings/asia/2018/asia-18-WANG-KSMA-Breaking-Android-kernel-isolation-and-Rooting-with-ARM-MMU-features.pdf

[2] https://github.com/2freeman/Slides/blob/main/PoC-2020-Three%20Dark%20clouds%20over%20the%20Android%20kernel.pdf

[3] https://www.longterm.io/cve-2020-0423.html

[4] https://i.blackhat.com/USA21/Wednesday-Handouts/us-21-Typhoon-Mangkhut-One-Click-Remote-Universal-Root-Formed-With-Two-Vulnerabilities.pdf

[5] https://conference.hitb.org/hitbsecconf2021sin/materials/D1T1%20-%20%20The%20Art%20of%20Exploiting%20UAF%20by%20Ret2bpf%20in%20Android%20Kernel%20-%20Xingyu%20Jin%20&%20Richard%20Neal.pdf

[6] https://community.arm.com/arm-community-blogs/b/graphics-gaming-and-vr-blog/posts/memory-management-on-embedded-graphics-processors

[7] https://community.arm.com/arm-community-blogs/b/graphics-gaming-and-vr-blog/posts/new-suite-of-arm-mali-gpus

[8] https://en.wikipedia.org/wiki/OpenCL

[9] https://gitlab.freedesktop.org/panfrost

[10]https://github.blog/2022-07-27-corrupting-memory-without-memory-corruption

[11] https://www.blackhat.com/asia-23/briefings/schedule/index.html#two-bugs-with-one-poc-rooting-pixel--from-android--to-android--30148

Thank you!

WANG, YONG (@ThomasKing2014)
ThomasKingNew@gmail.com