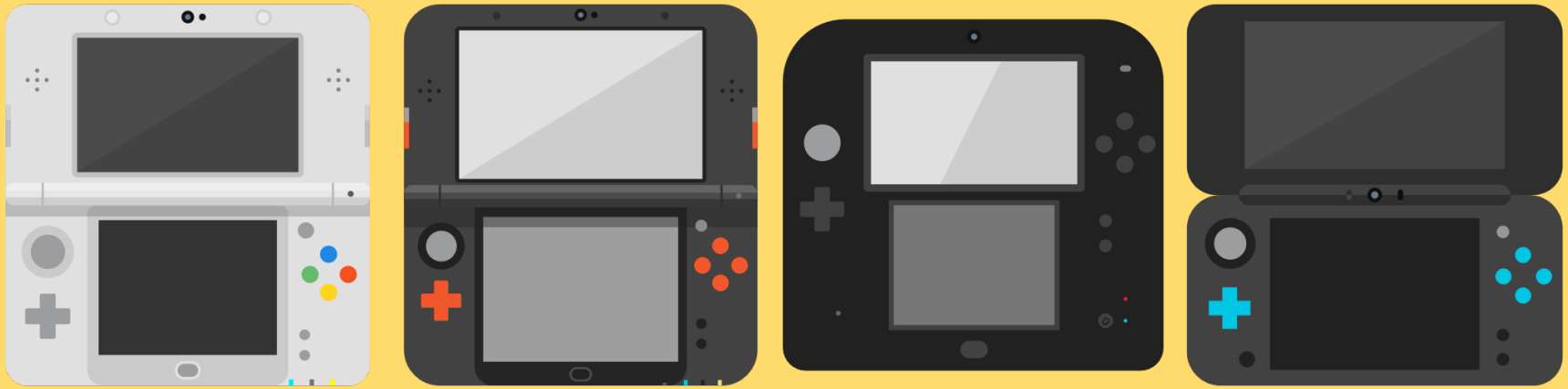


# Jailbreaking the 3DS

@smealum

# Intro to 3DS



## ”Old” 3DS line



3DS



3DS XL




2DS

- Out starting 2011
- CPU: 2x ARM11 MPCore (268MHz)
- GPU: DMP PICA
- RAM: 128MB FCRAM, 6MB VRAM
- IO/Security CPU: ARM946
- Hardware-based backwards compatible with DS games
- Fully custom microkernel-based OS

## New 3DS line



New 3DS

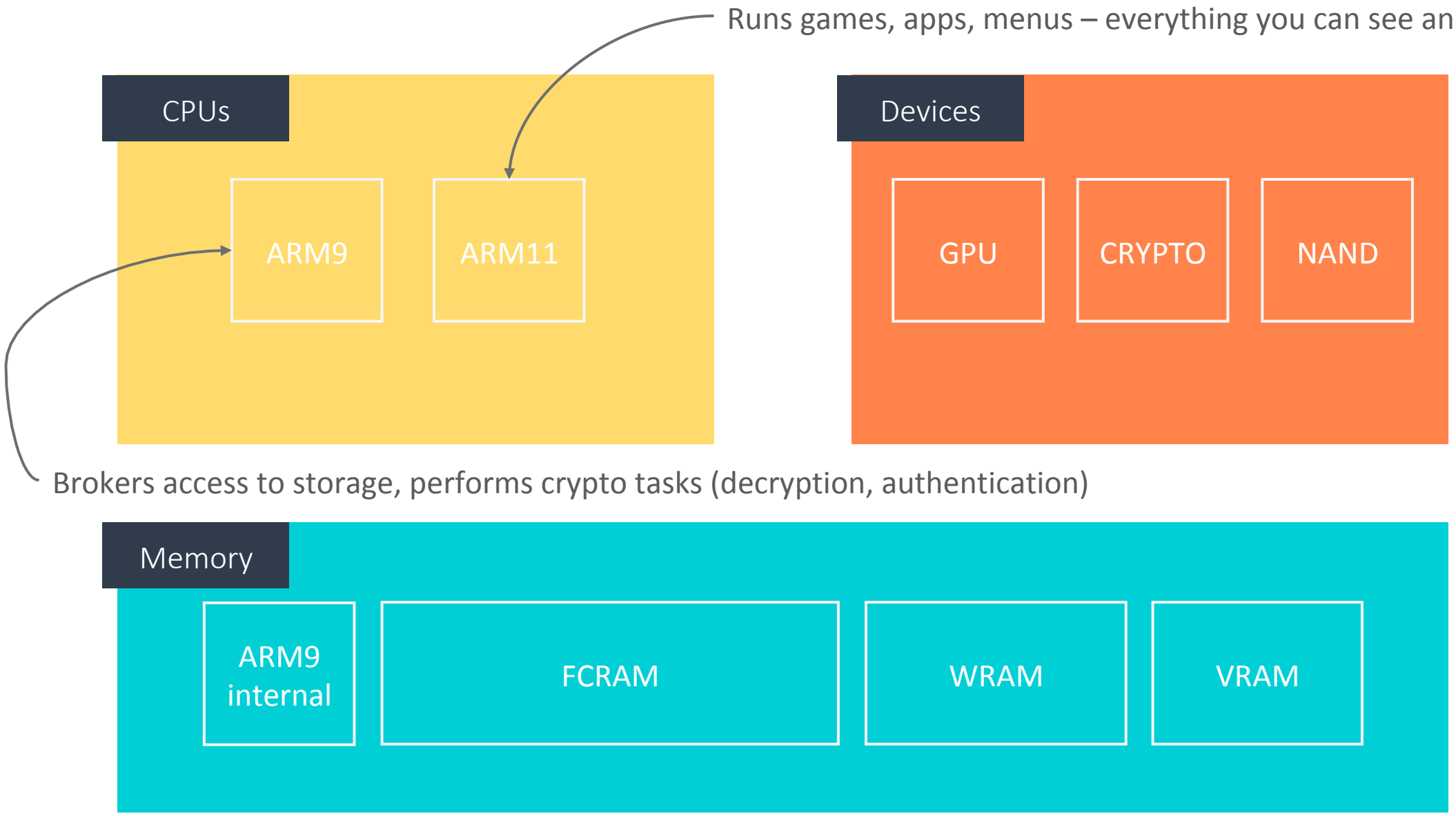


New 3DS XL

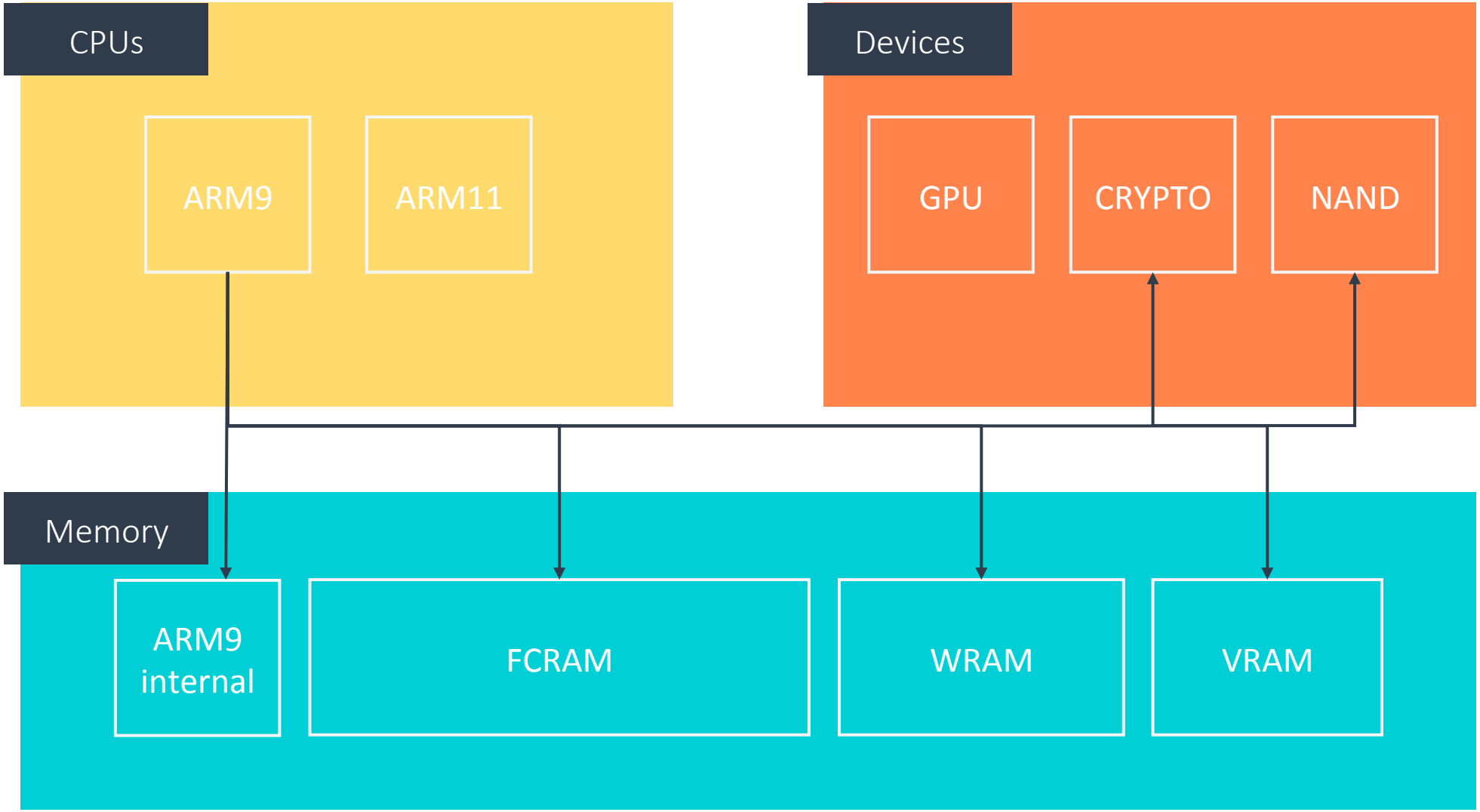


New 2DS XL

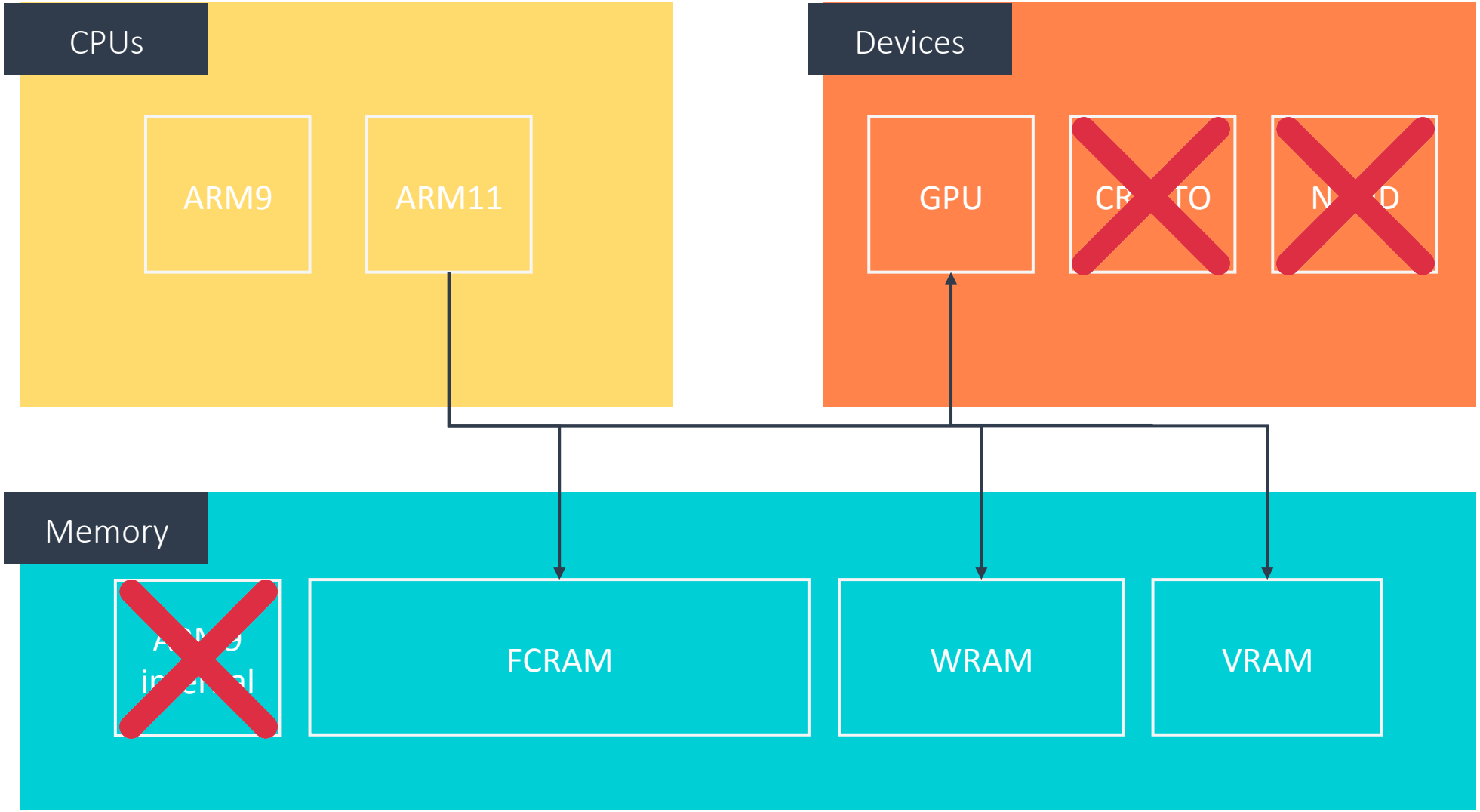
- Out starting 2014
- CPU: 4x ARM11 MPCore (804MHz)
- GPU: DMP PICA
- RAM: 256MB FCRAM, 6MB VRAM
- IO/Security CPU: ARM946
- Hardware-based backwards compatible with DS games
- Fully custom microkernel-based OS



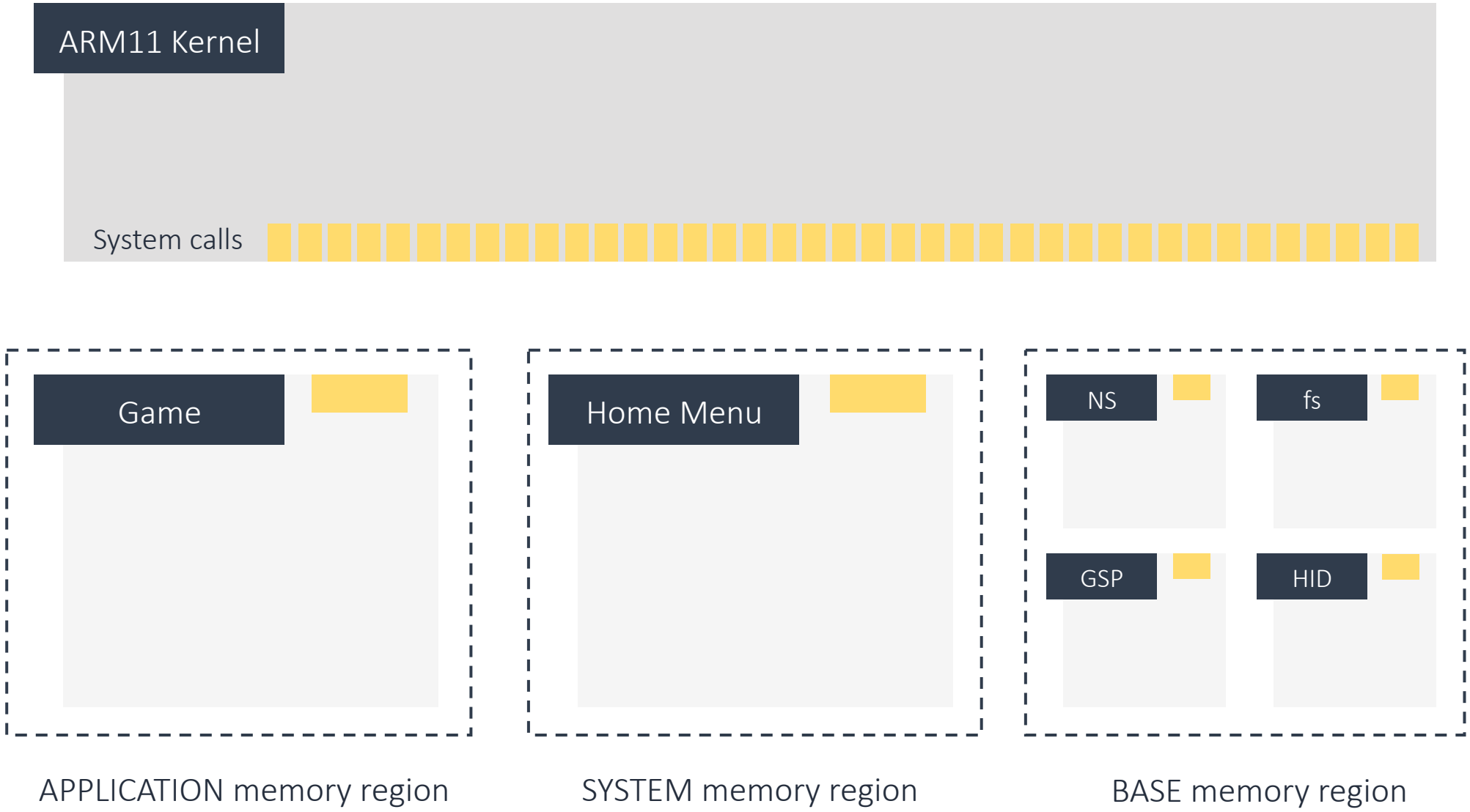

# 3DS hardware overview



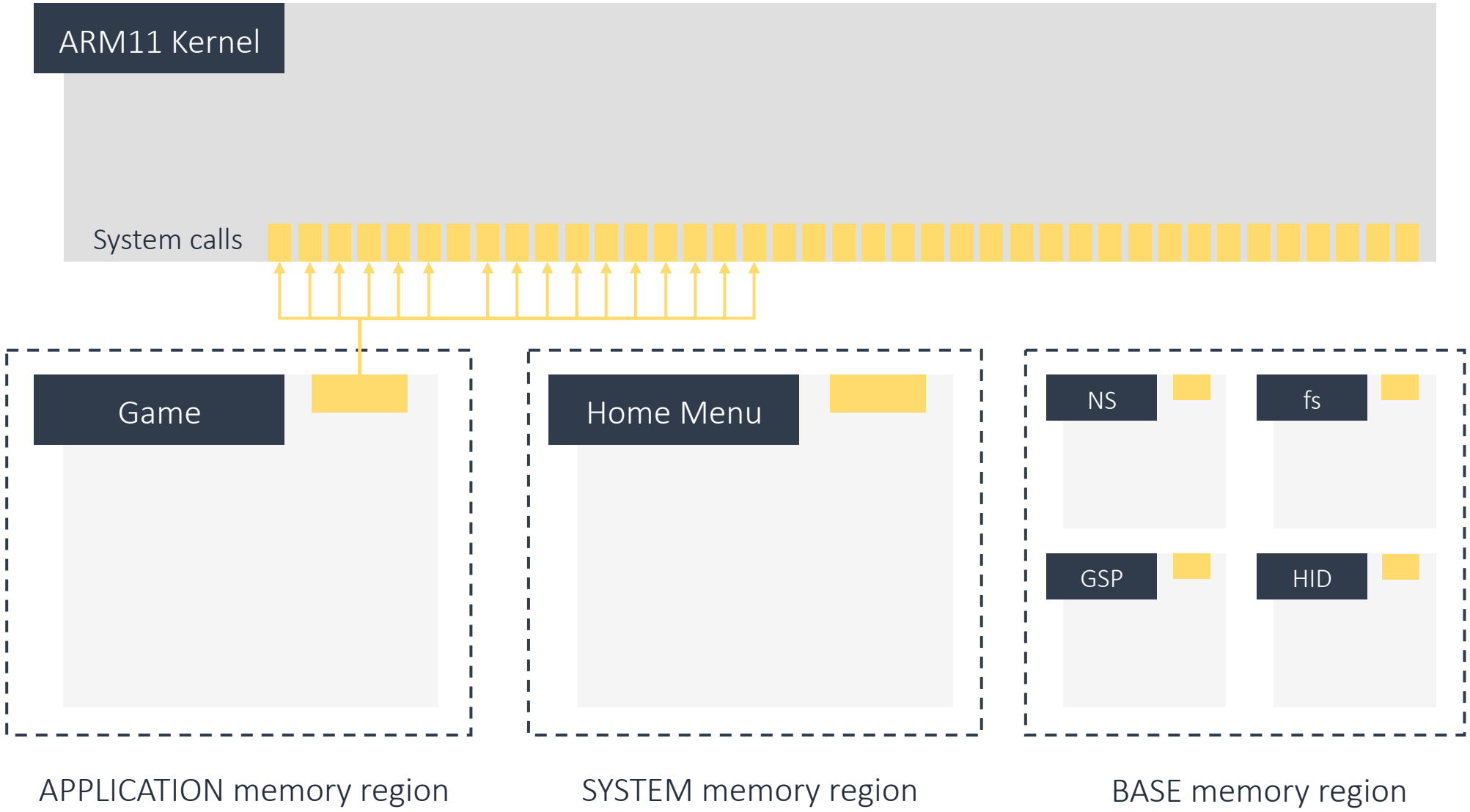

ARM9: access to almost everything



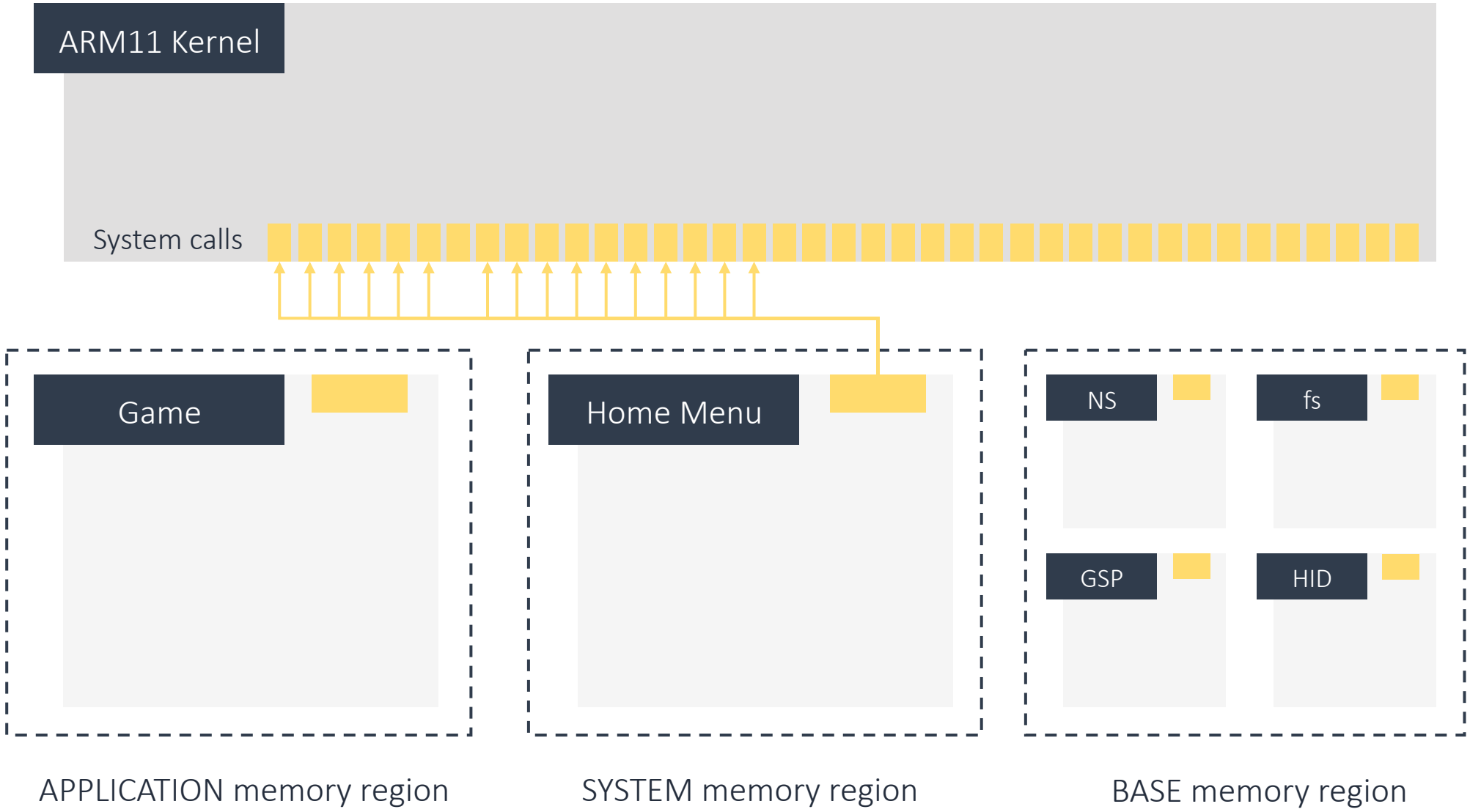

ARM11: more limited access to hardware



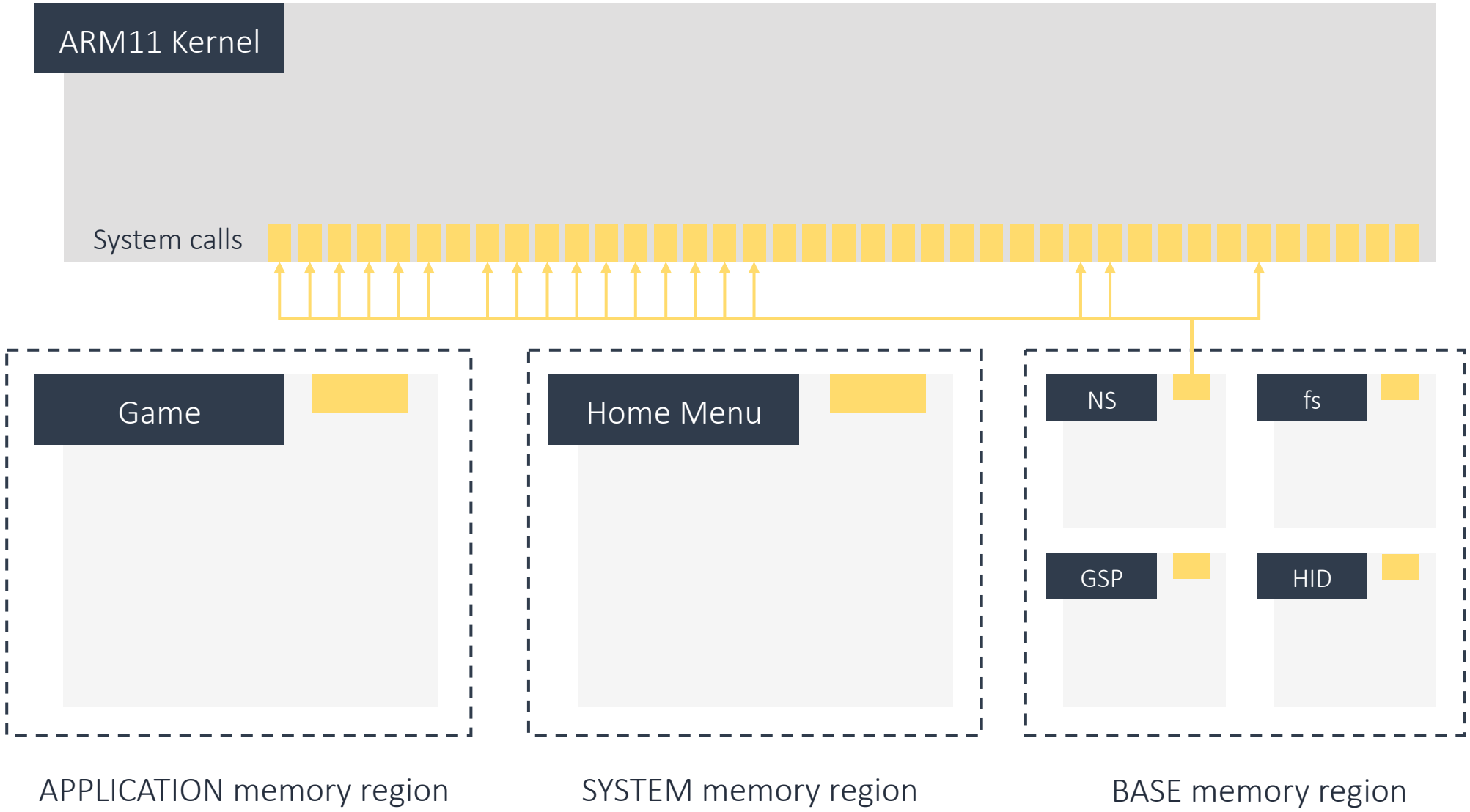

# The ARM11's microkernel architecture



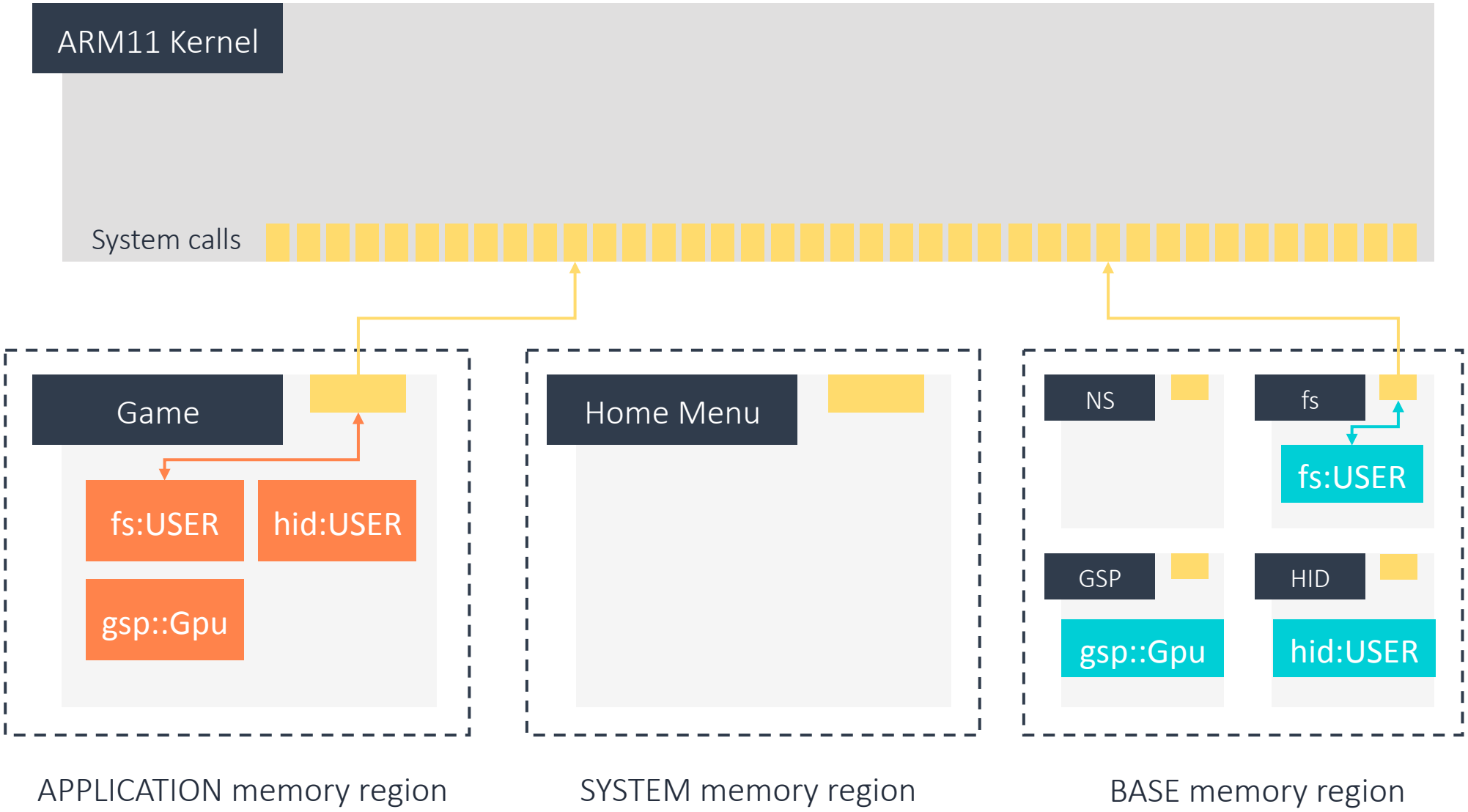

# The ARM11's microkernel architecture: system call access control



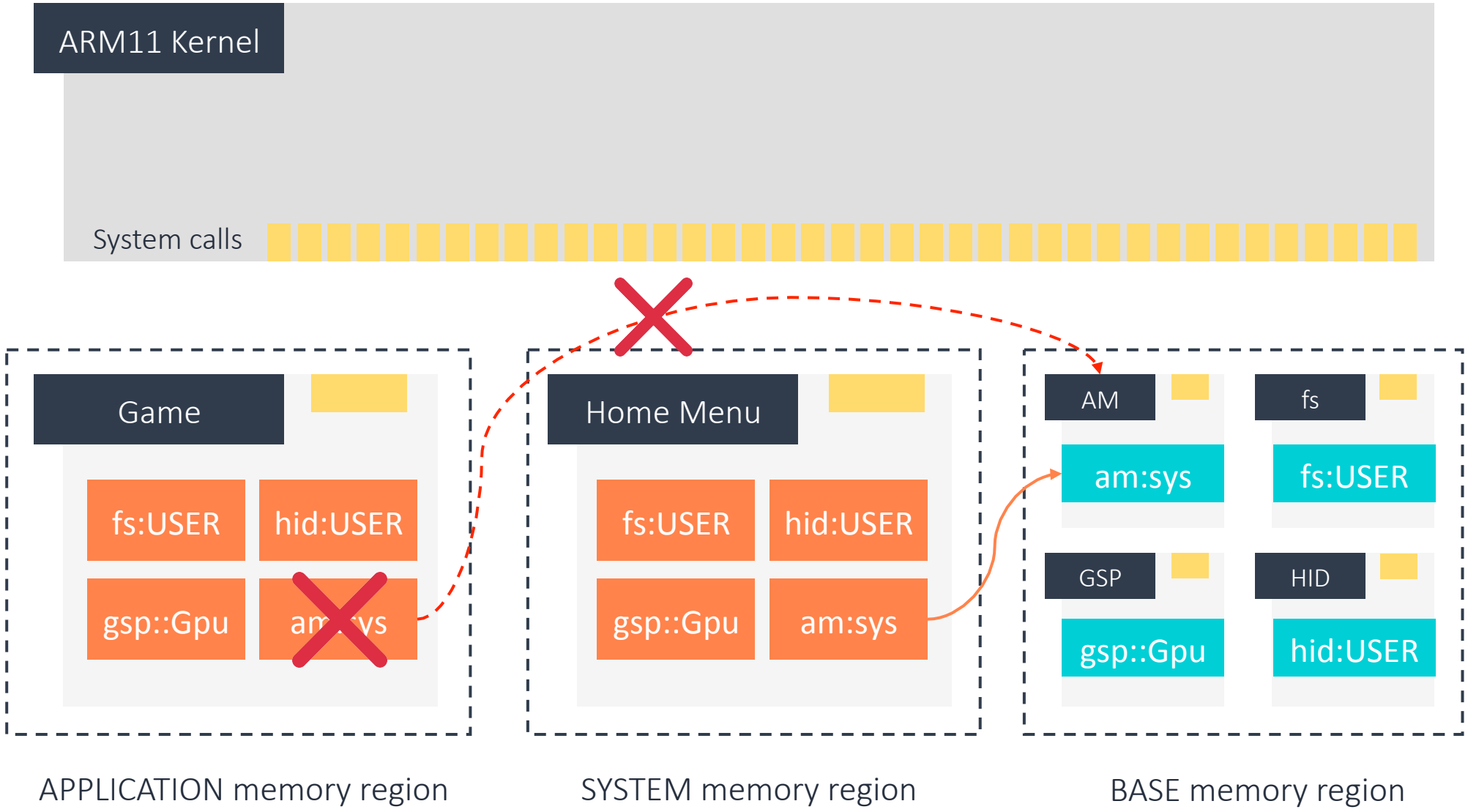

# The ARM11's microkernel architecture: system call access control



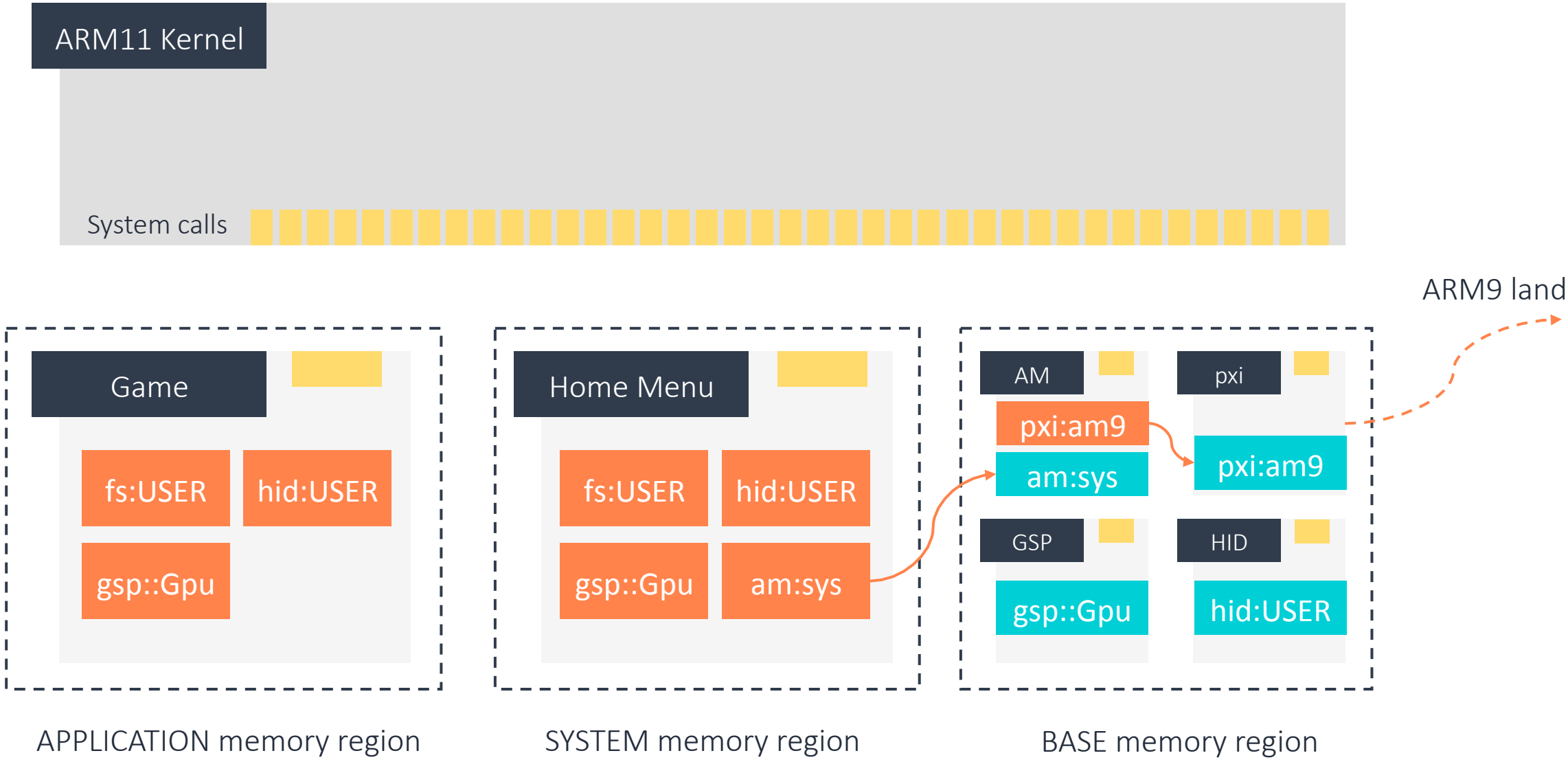

# The ARM11's microkernel architecture: system call access control



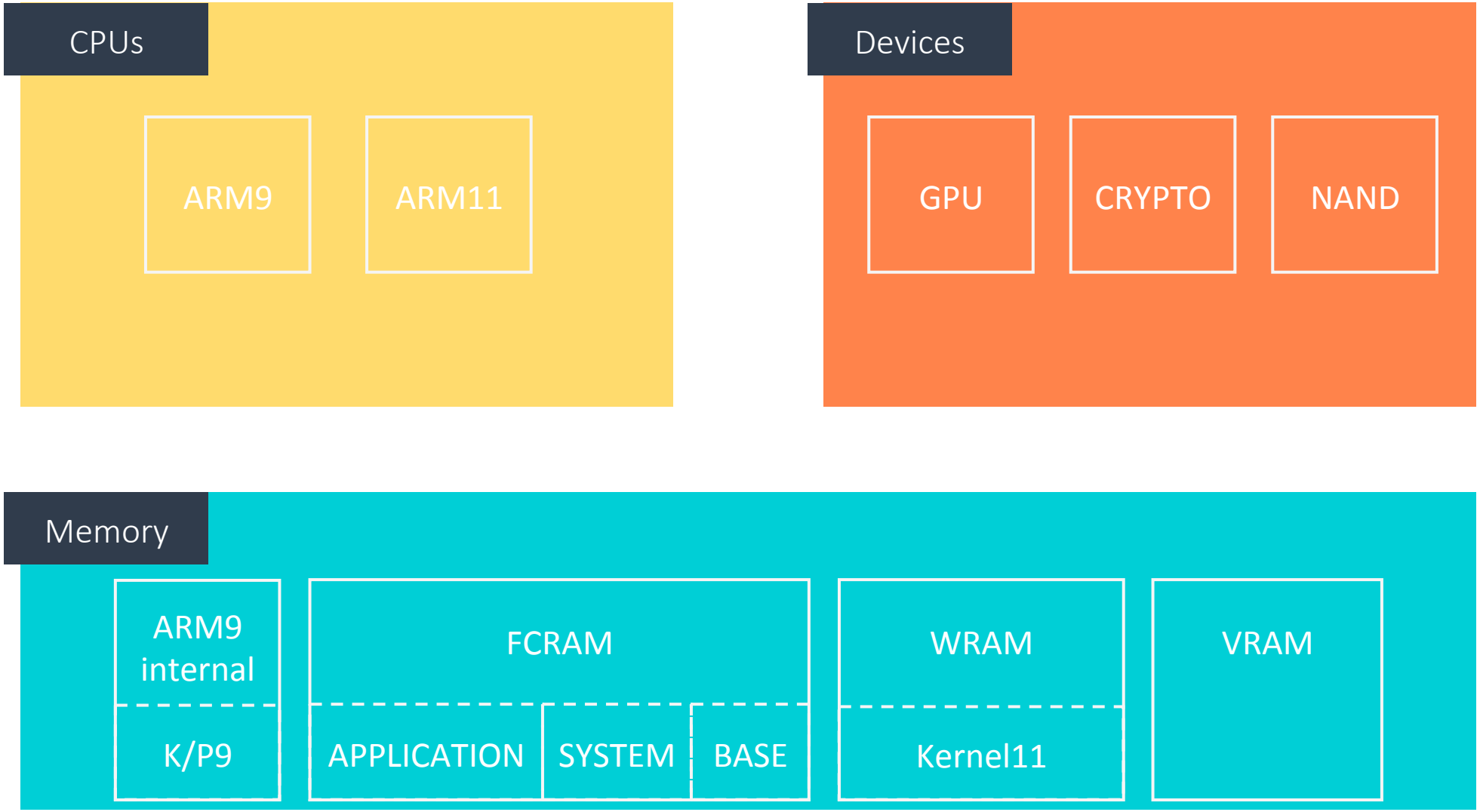

# The ARM11's microkernel architecture: services




# The ARM11's microkernel architecture: service access control




# The ARM11's microkernel architecture: service access control




Physical memory region separation

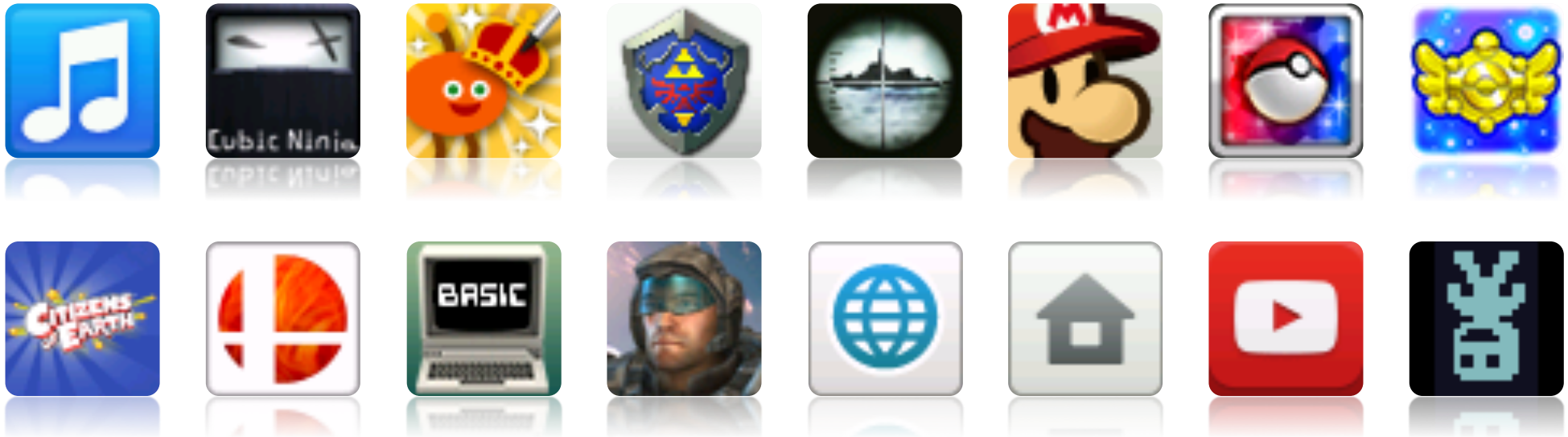
# Anatomy of a 3DS “jailbreak”

1. Compromise a user-mode game or app
2. Escalate privilege to expand attack surface
3. Compromise ARM11 kernel
4. Compromise ARM9



# Getting code execution

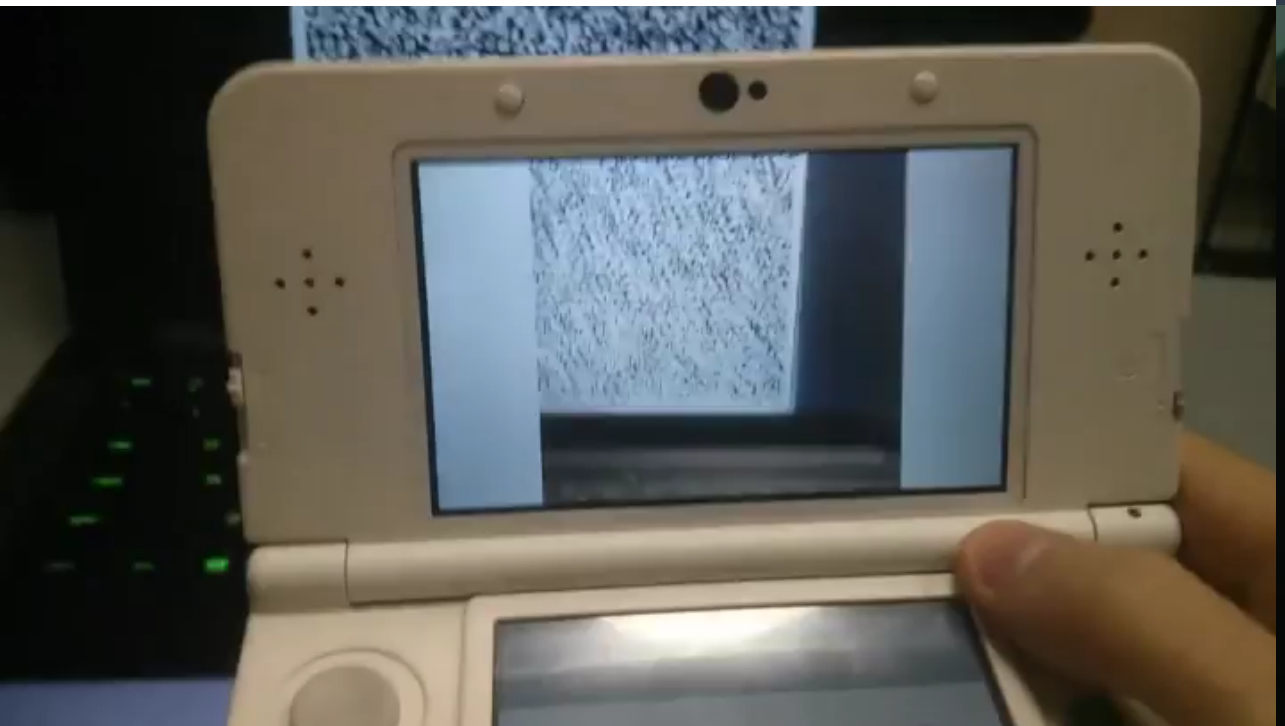
Where do we start?




Previous 3DS entrypoints

# Cubic Ninja

- Vulnerable custom level parser
- Levels shareable over QR codes…
- No ASLR, no stack cookies etc. makes file format bugs fair game

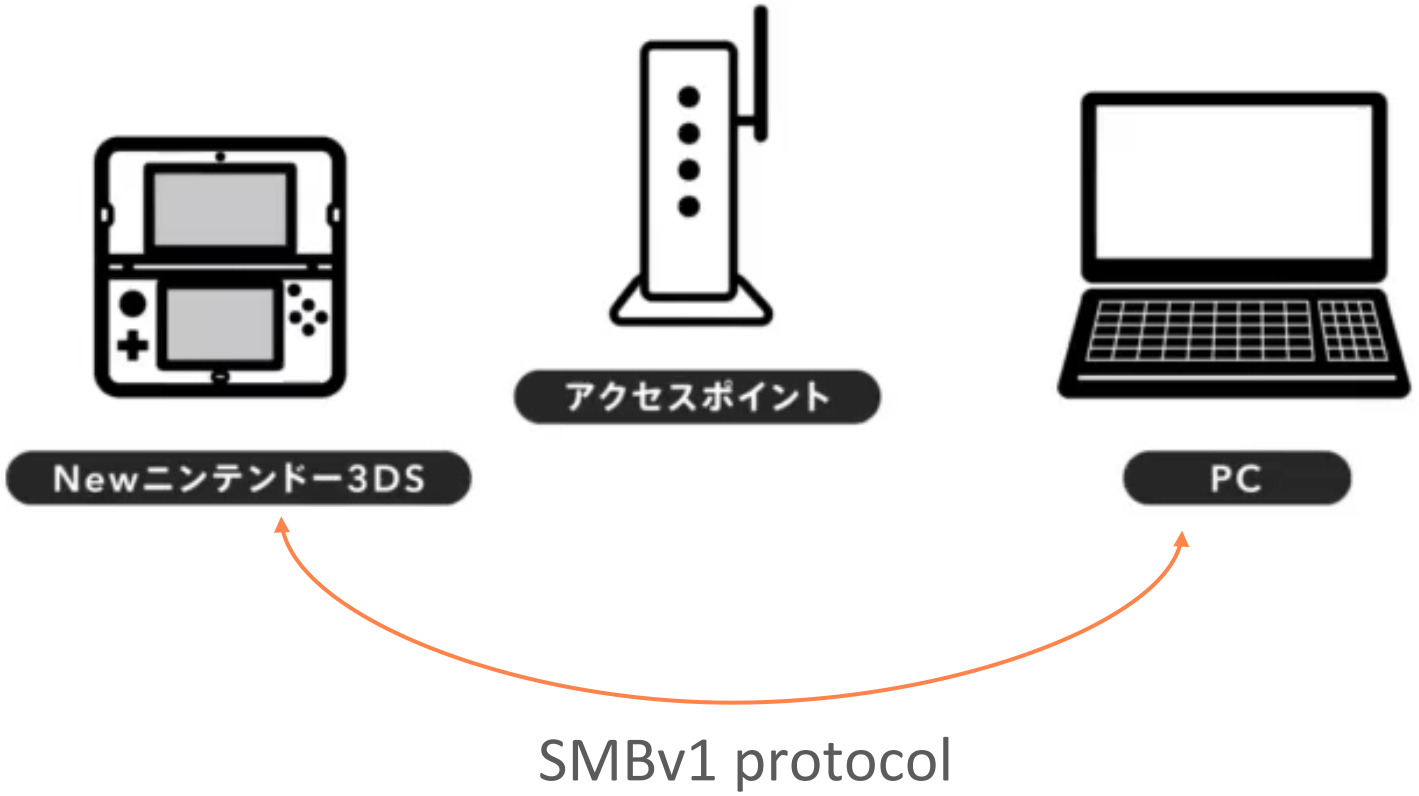


# Web Browsers

- 3DS has a built-in browser
- ASLR, stack cookies etc. have never stopped a browser exploit
- Nintendo’s threat model should assume compromised user-mode



# mcopy

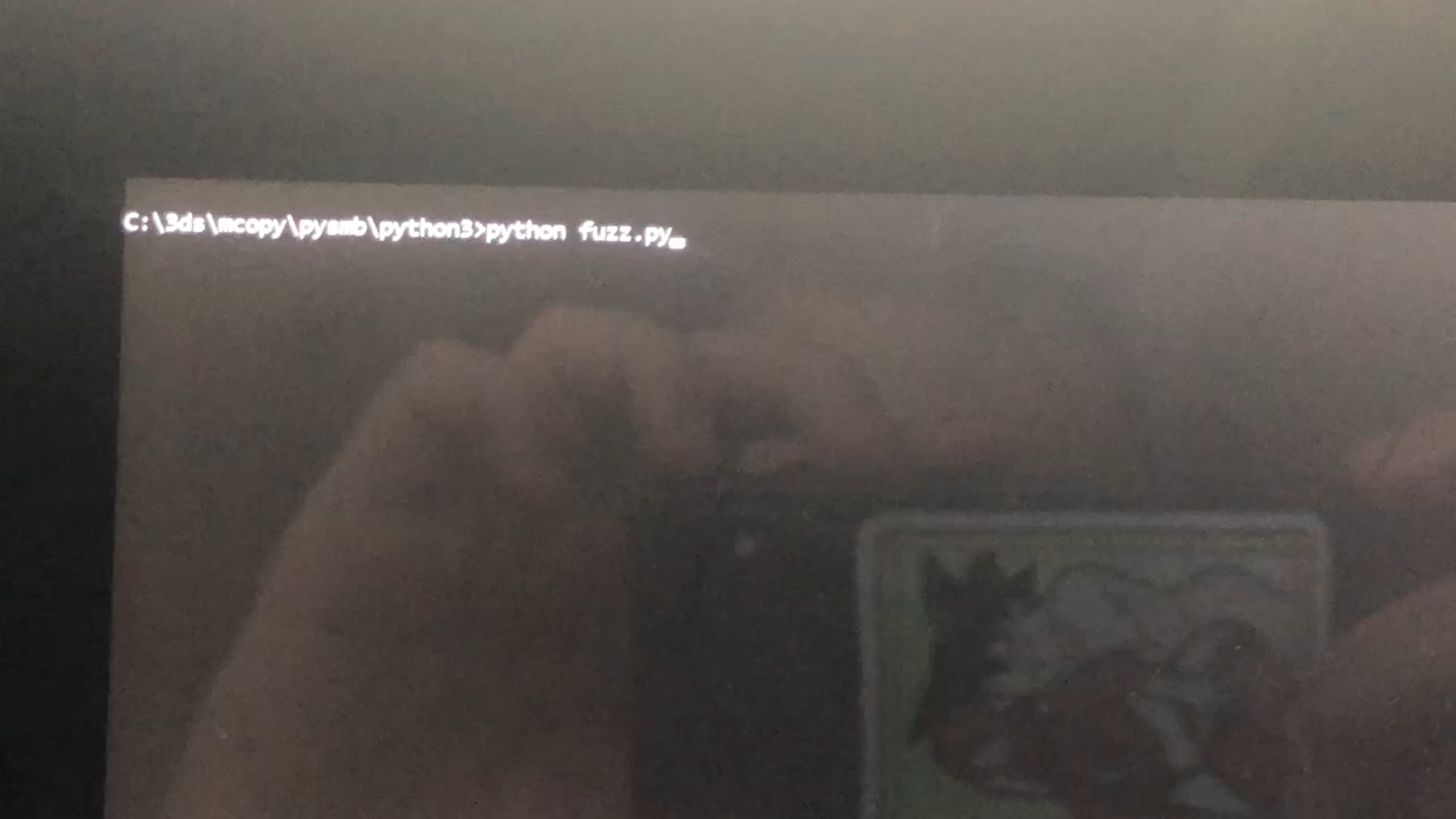


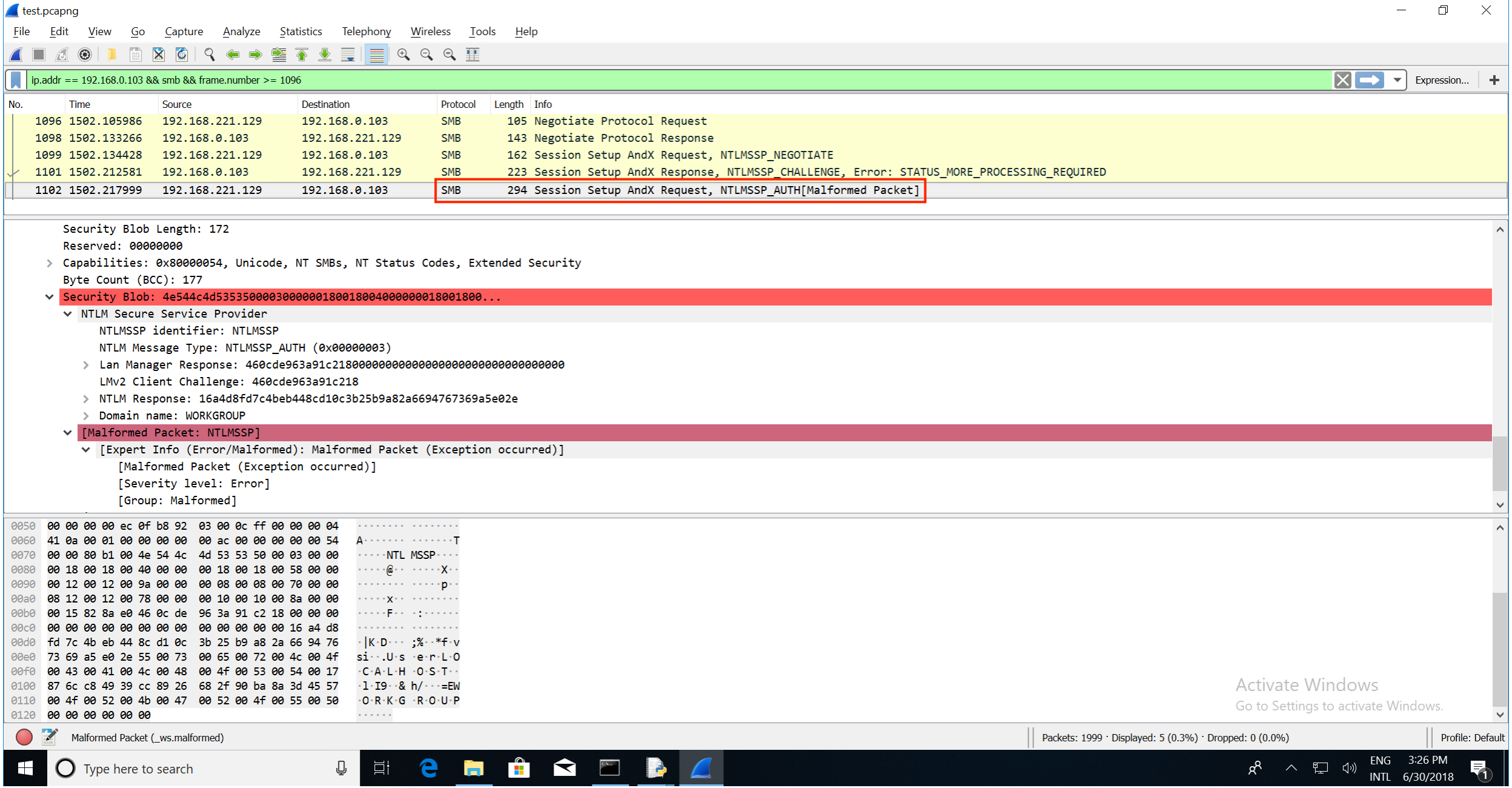
# Attacking mcopy

- Bug finding strategy: fuzzing
- Used [github.com/miketeo/pysmb](github.com/miketeo/pysmb)
- Had to adjust some code to make it compatible with mcopy
- Added 6 lines of fuzzing code…

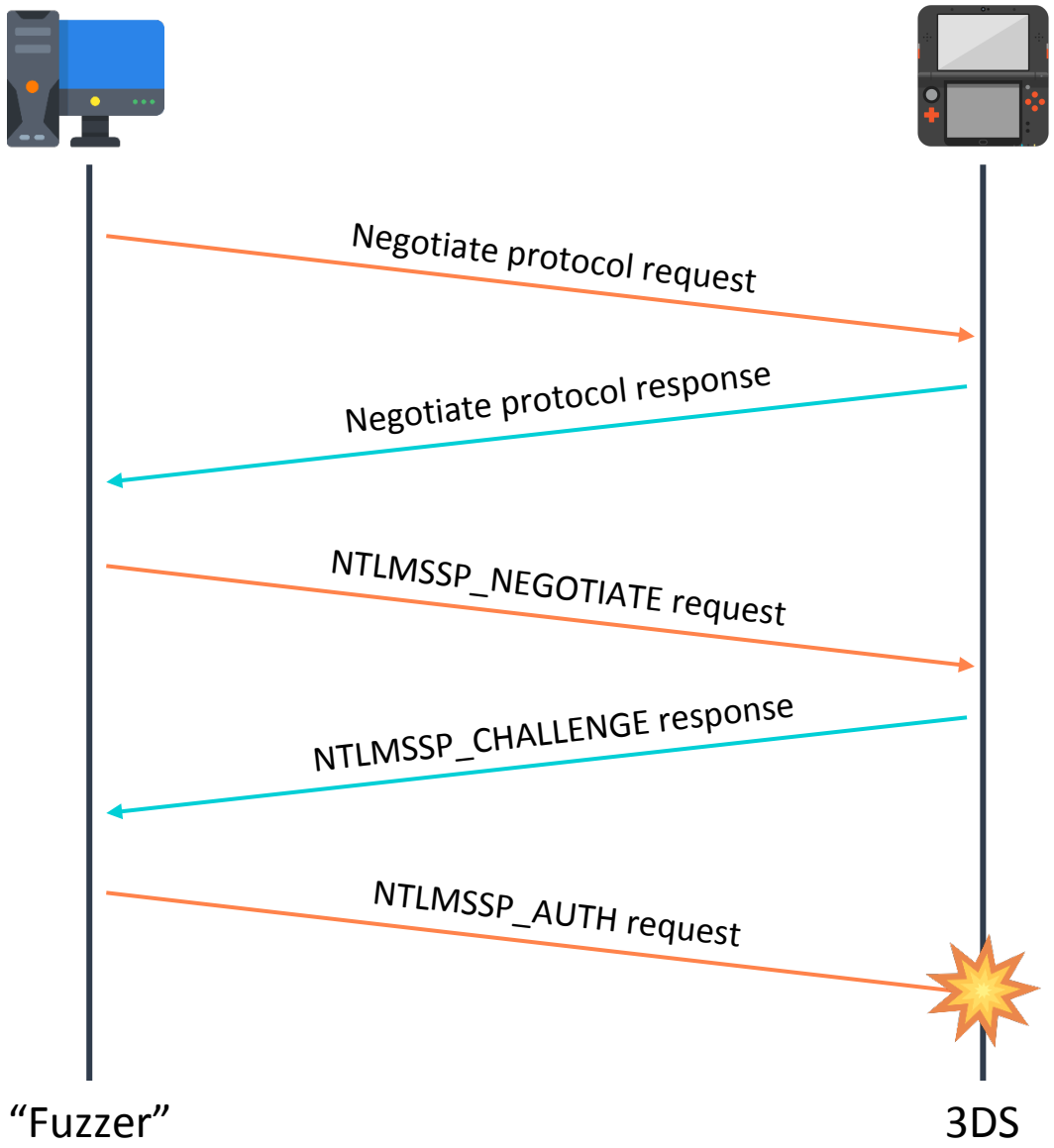
```
def _sendSMBMessage_SMB1(self, smb_message):
    ...
        input = smb_message.encode()
        output = []

        # TMP FUZZ TEST
        for i in range(len(input)):
            val = input[i]
            if randint(1, 1000) < FUZZ_thresh:
                mask = (1 << randint(0, 7))
                val ^= mask
            output += [val]
        # END TMP FUZZ TEST

        smb_message.raw_data = bytes(output)
    ...
    self.sendNMBMessage(smb_message.raw_data)
```

The worst SMB fuzzer ever written

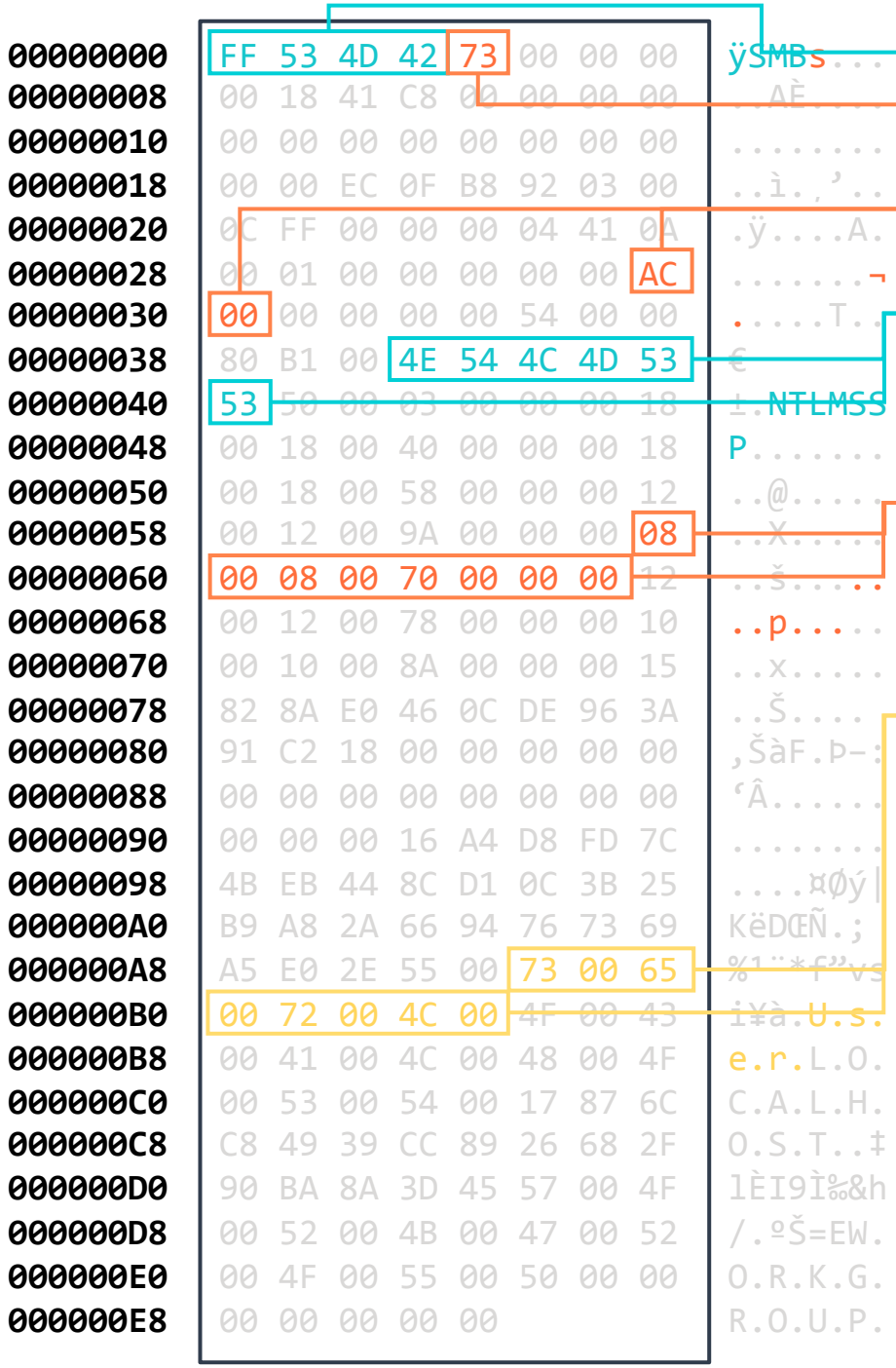

C:\3ds\mcopy\pysmb\python3>python fuzz.py




# Initial fuzz crash: Wireshark trace




# Initial fuzz crash: protocol diagram




0x0000: SMB magic number

0x0004: SMB command (0x73: Sessions Setup AndX)

...

0x002F: Security blob size (0xAC bytes)

...

0x003B: Security blob magic number

...

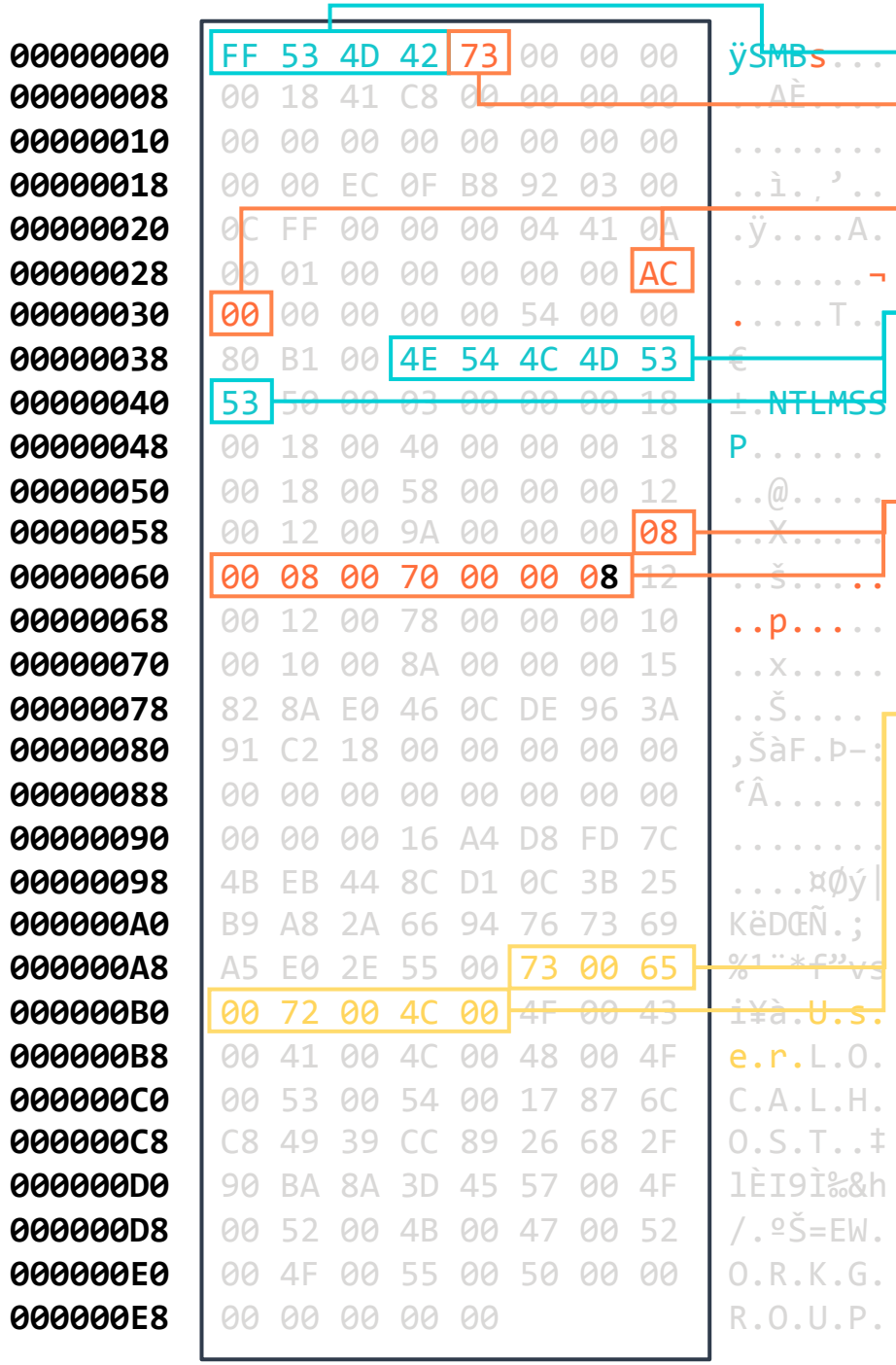
0x005F: Username data descriptor

- 0x00: Data length (0x08)
- 0x02: Maximum data length (0x08)
- 0x04: Data offset (0x00000070)

...

0x00AB: Username data (“User”)

# Initial fuzz crash: normal packet




0x0000: SMB magic number

0x0004: SMB command (0x73: Sessions Setup AndX)

...

0x002F: Security blob size (0xAC bytes)

...

0x003B: Security blob magic number

...

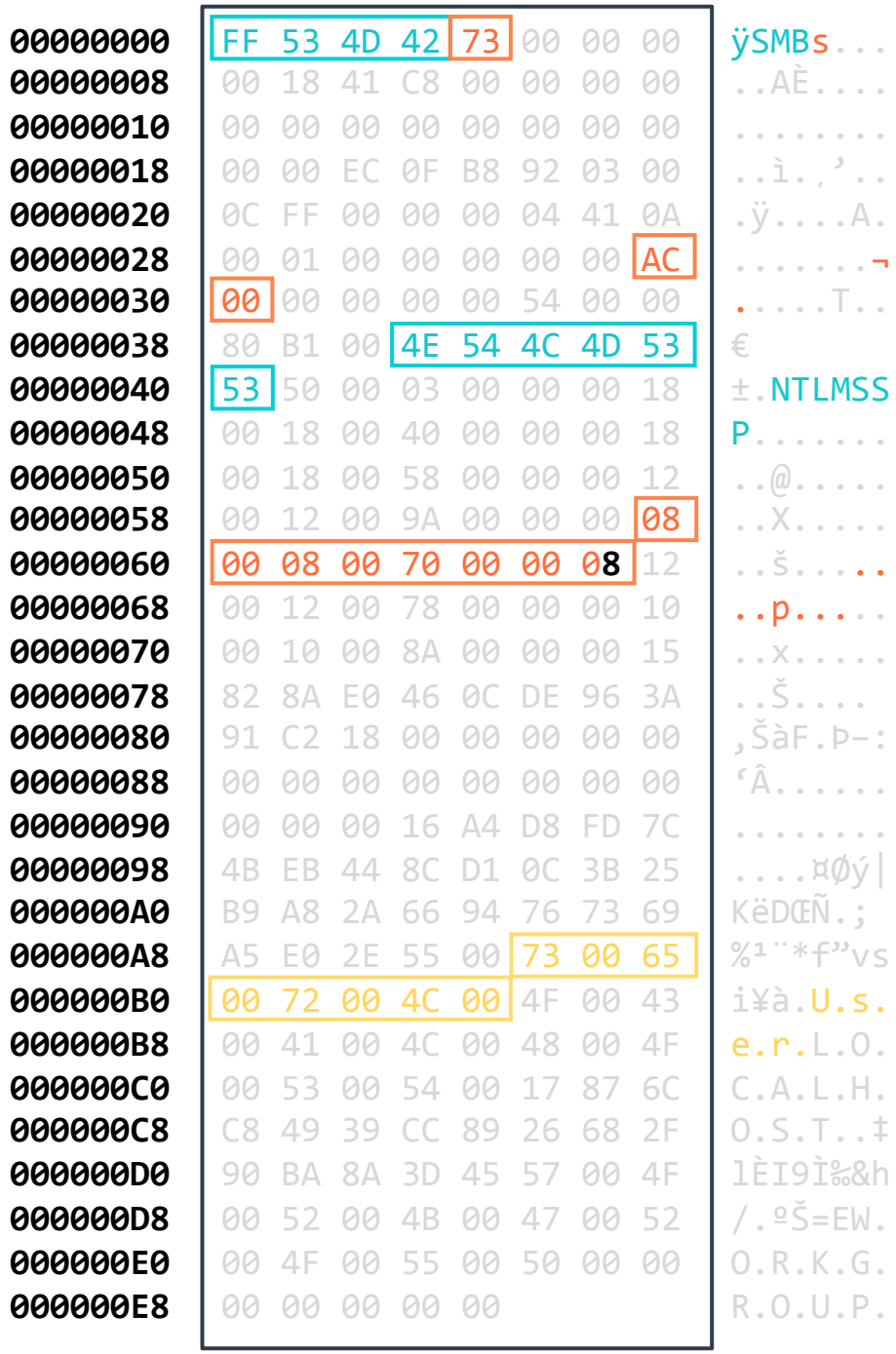
0x005F: Username data descriptor

- 0x00: Data length (0x08)
- 0x02: Maximum data length (0x08)
- 0x04: Data offset (0x08000070)

...

0x00AB: Username data (“User”)

# Initial fuzz crash: corrupted packet




**Processor:** ARM11 (core 0)
**Exception type:** data abort
**Fault status:** Translation - Section
**Current process:** mcopy (0004001000024100)

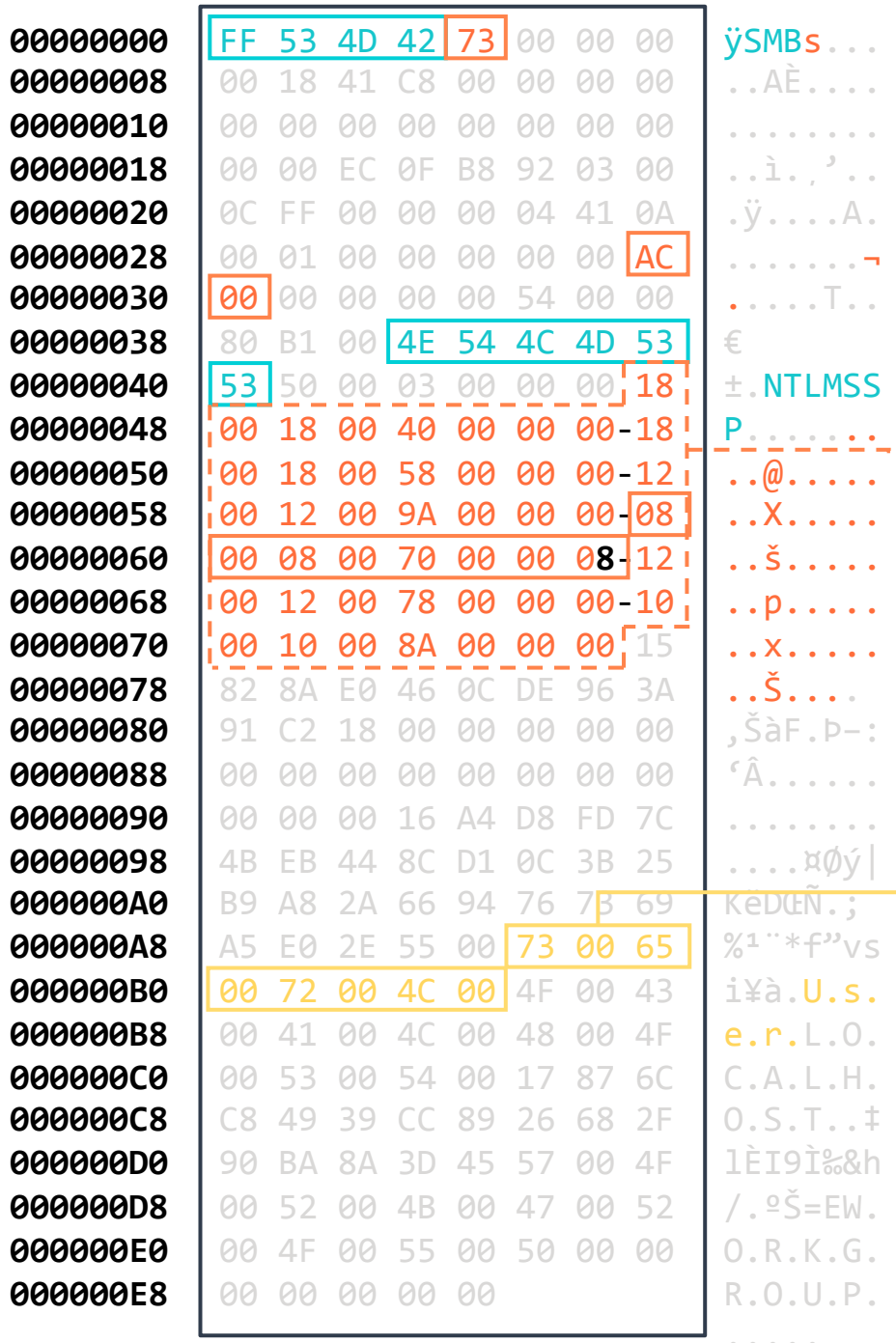
**Register dump:**

| | | | |
|---|---|---|---|
| r0 | 0885b858 | r1 | **1085b724** |
| r2 | fffffe8 | r3 | 00000000 |
| r4 | 08002d60 | r5 | 00000082 |
| r6 | 0885b6b4 | r7 | 000000ac |
| r8 | 00000070 | r9 | 00000000 |
| r10 | 00000002 | r11 | 00000004 |
| r12 | 80000000 | sp | 08002d28 |
| lr | 00194b44 | pc | 00164e84 |
| cpsr | 80000010 | dfsr | 00000005 |
| ifsr | 0000100b | far | **1085b724** |
| fpexc | 00000000 | fpinst | eebc0ac0 |
| fpinst2 | eebc0ac0 | | |
| FAR | 1085b724 | **Access type:** Read | |

**Crashing instruction:**

```
ldmmi r1!, {r3, r4}
```

# Initial fuzz crash: exception dump




```
int SecurityBlob::parse(u8* buffer, int length)
{
  int result = -1;
  if ( security_blob_len >= 0x58 )
  {
    int offset = this->unpack_ntlmssp_header(buffer, length);

    if ( offset >= 0 )
    {
      for(int i = 0; i < 6; i++)
      {
        offset += this->unpack_length_offset(buffer + offset,
                                             &this->fields[i]);
      }

      offset += this->parse_negociate_flags(buffer + offset);

      int username_length = this->fields[3].length;
      if ( username_length && username_length <= length - offset )
      {
        this->username_buffer = malloc(username_length & 0xFFFFFFFE);
        memmove(this->username_buffer,
                buffer + this->fields[3].offset,
                username_length);
        offset += username_length;
      }

       ...

    }
  }
  return result;
}
```

# Initial fuzz crash: code

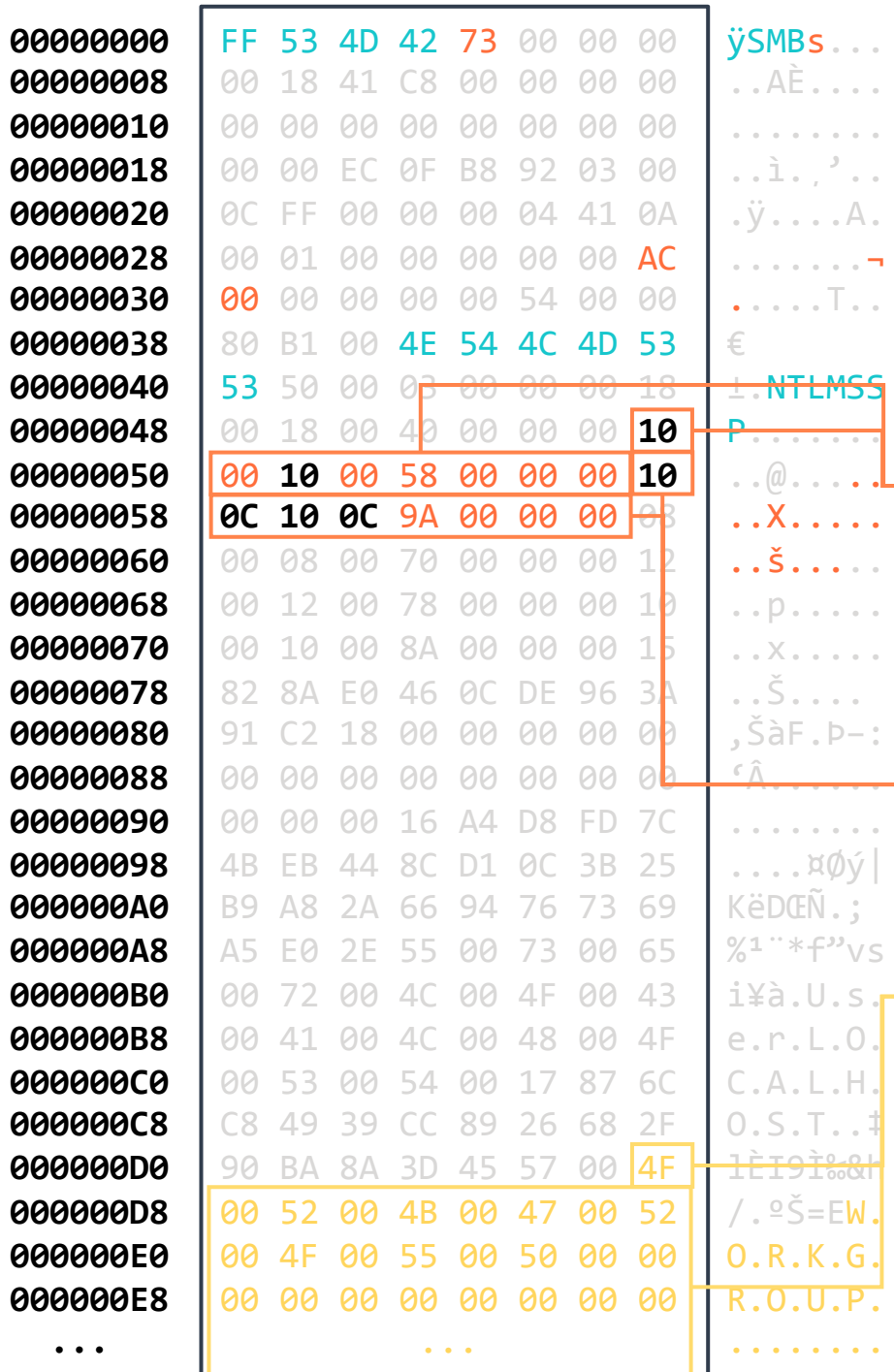
```
int sub_1910C4()
{
  ...
  secblob->parse(buffer, length);

  if(secblob->fields[1].length != 0x18)
    secblob->sub_18EA84(0x18, ...);
  else
    secblob->sub_18EBB0(secblob->fields[1].length, ...);
  ...
}
```

```
int SecurityBlob::parse(u8* buffer, int length)
{
  ...
  for(int i = 0; i < 6; i++)
  {
    offset += this->unpack_length_offset(buffer + offset,
                                         &this->field[i]);
  }
  ...
}
```

```
int SecurityBlob::sub_18EBB0(...)
{
  wchar_t local_buffer[0x20];

  ...

  memmove(local_buffer, this->domain_buffer, this->domain_length);

  ...
}
```



# Exploitable vuln: code




0x0000: SMB magic number
0x0004: SMB command (0x73: Session Setup AndX)
...
0x002F: Security blob size (0xAC bytes)
...
0x003B: Security blob magic number
...
0x0057: NTLM response data descriptor
  0x00: Data length (0x10 != 0x18)
  0x02: Maximum data length (0x10 != 0x18)
  0x04: Data offset (0x9A)

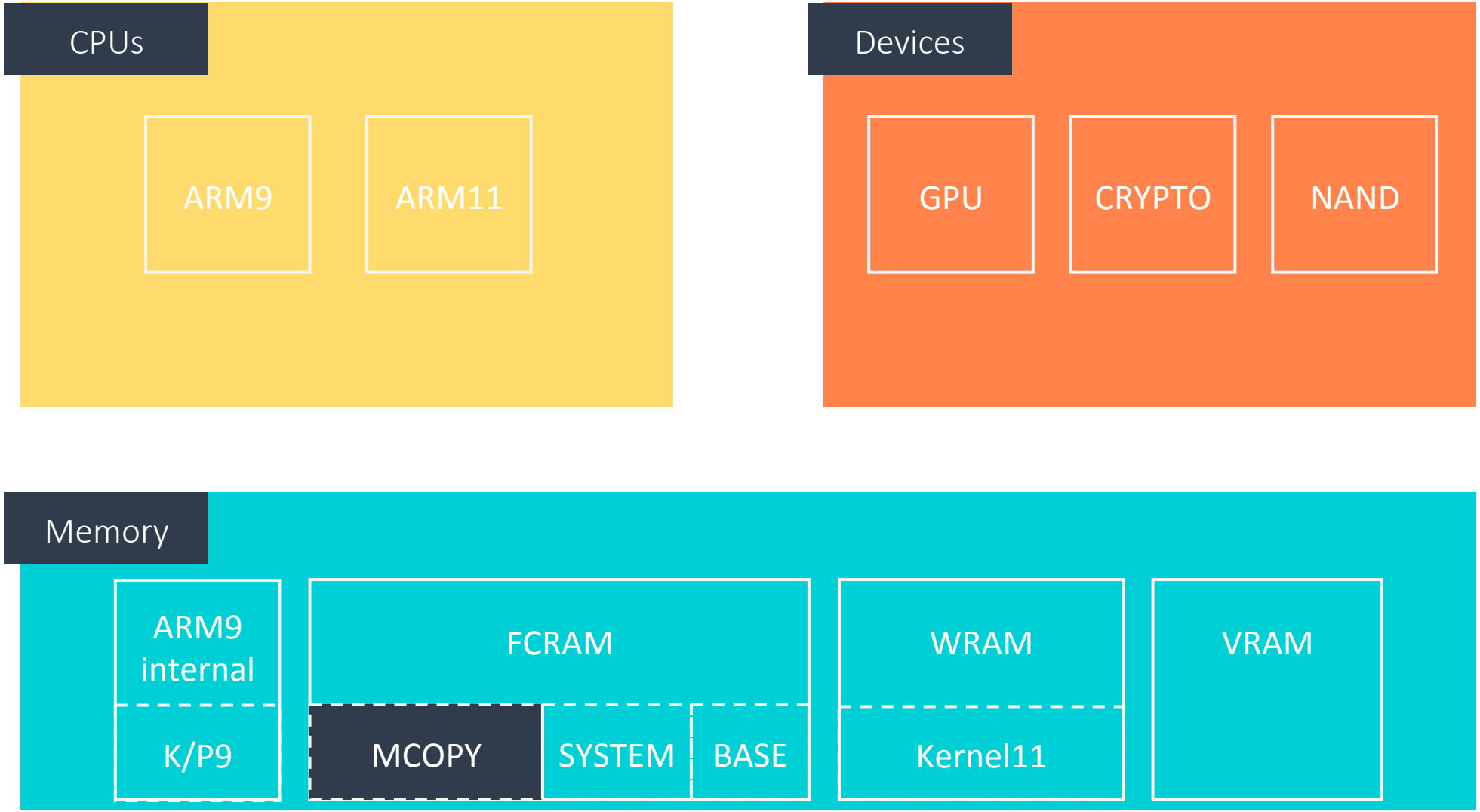
...
0x0057: Domain name data descriptor
  0x00: Data length (0xC10 > 0x20)
  0x02: Maximum data length (0xC10 > 0x20)
  0x04: Data offset (0x9A)

...
0x00AB: Stack smash overwrite payload (0xC10 bytes)

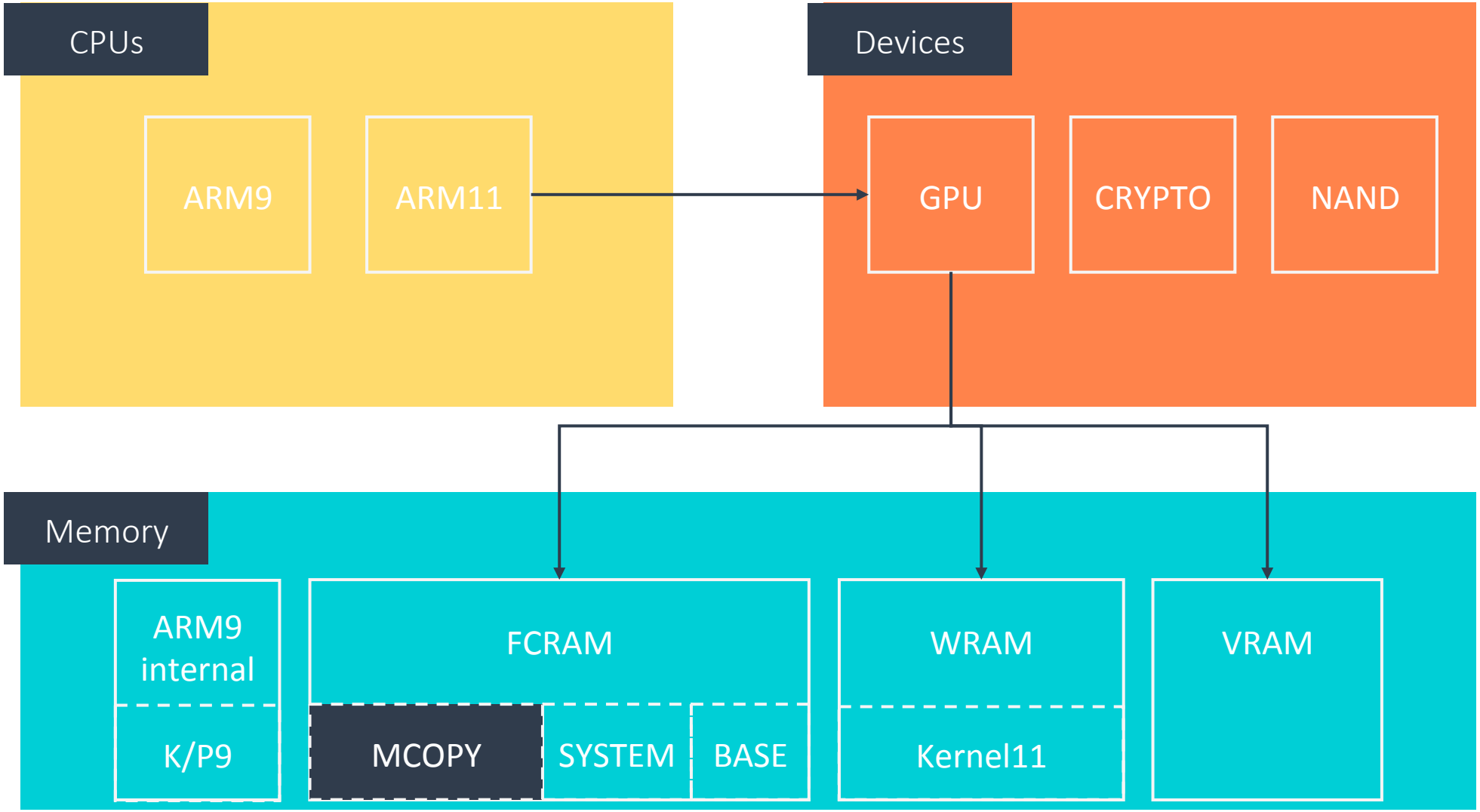
# Exploitable vuln: corrupted packet

# Code execution?

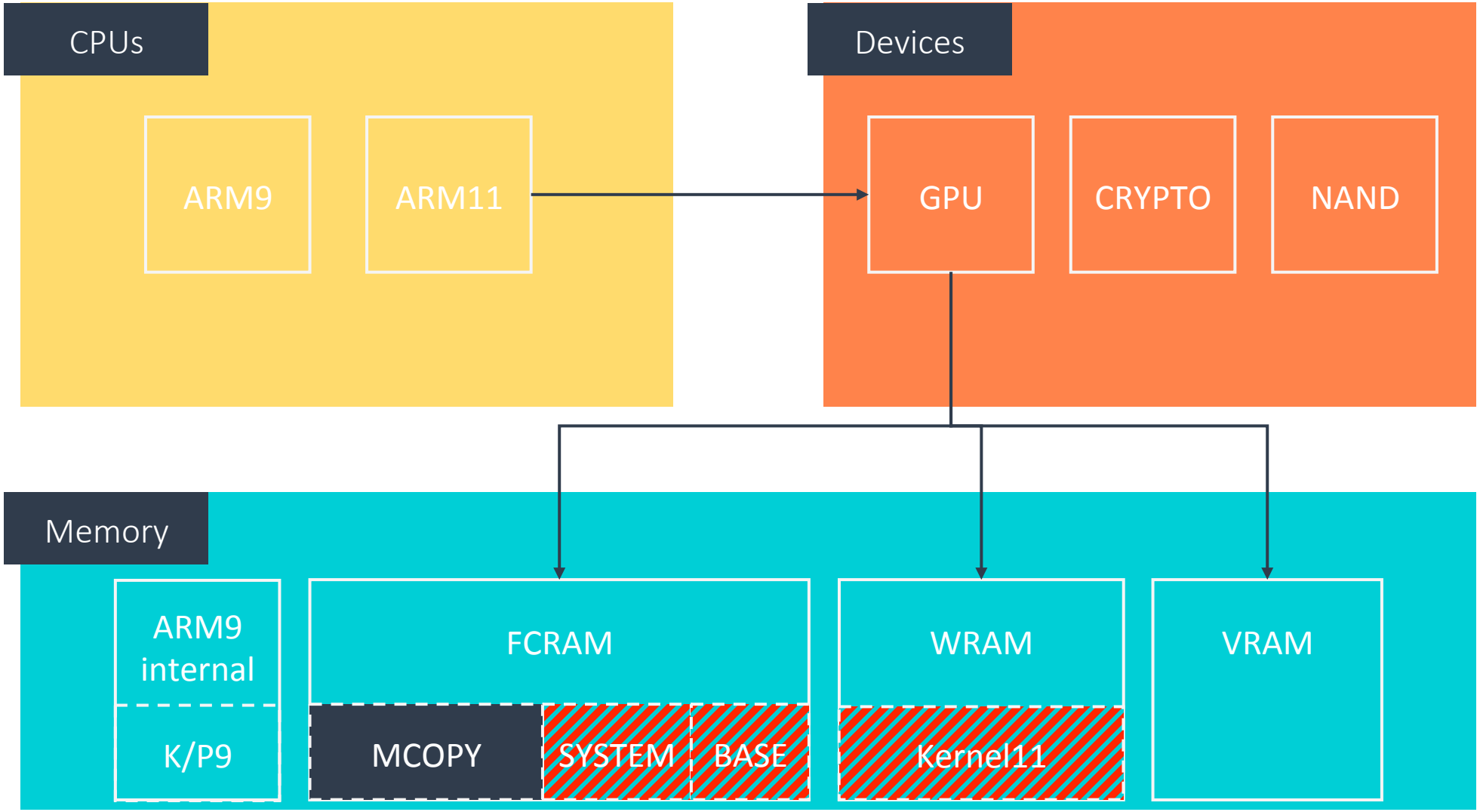
- Strict DEP – OS never allocates RWX memory in user-mode
- Only 2 system modules can create more executable memory
  - loader: loads main process binaries
  - ro: loads dynamic libraries (CROs – think DLL equivalent for 3DS)
- But what if we don’t need more executable memory…?



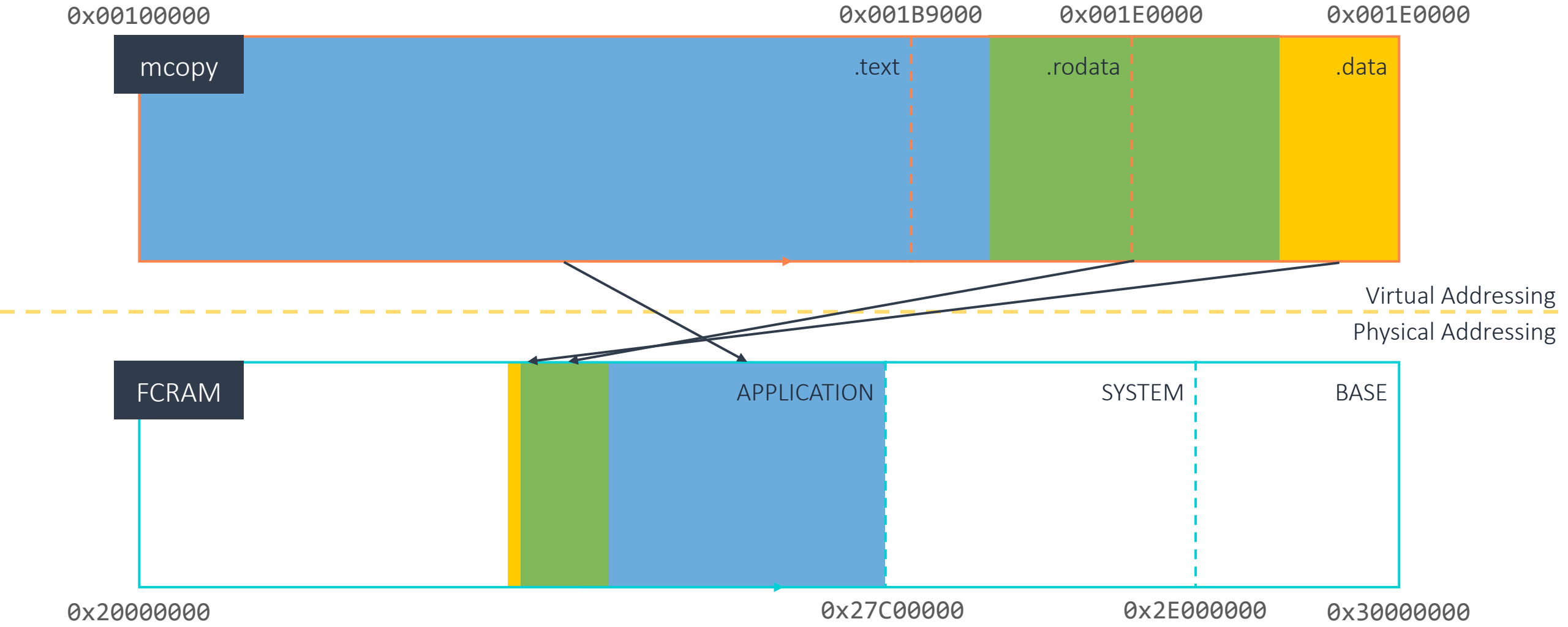

Physical memory region separation


CPUs
ARM9
ARM11
Devices
GPU
CRYPTO
NAND
Memory
ARM9 internal
K/P9
FCRAM
MCOPY
SYSTEM
BASE
WRAM
Kernel11
VRAM

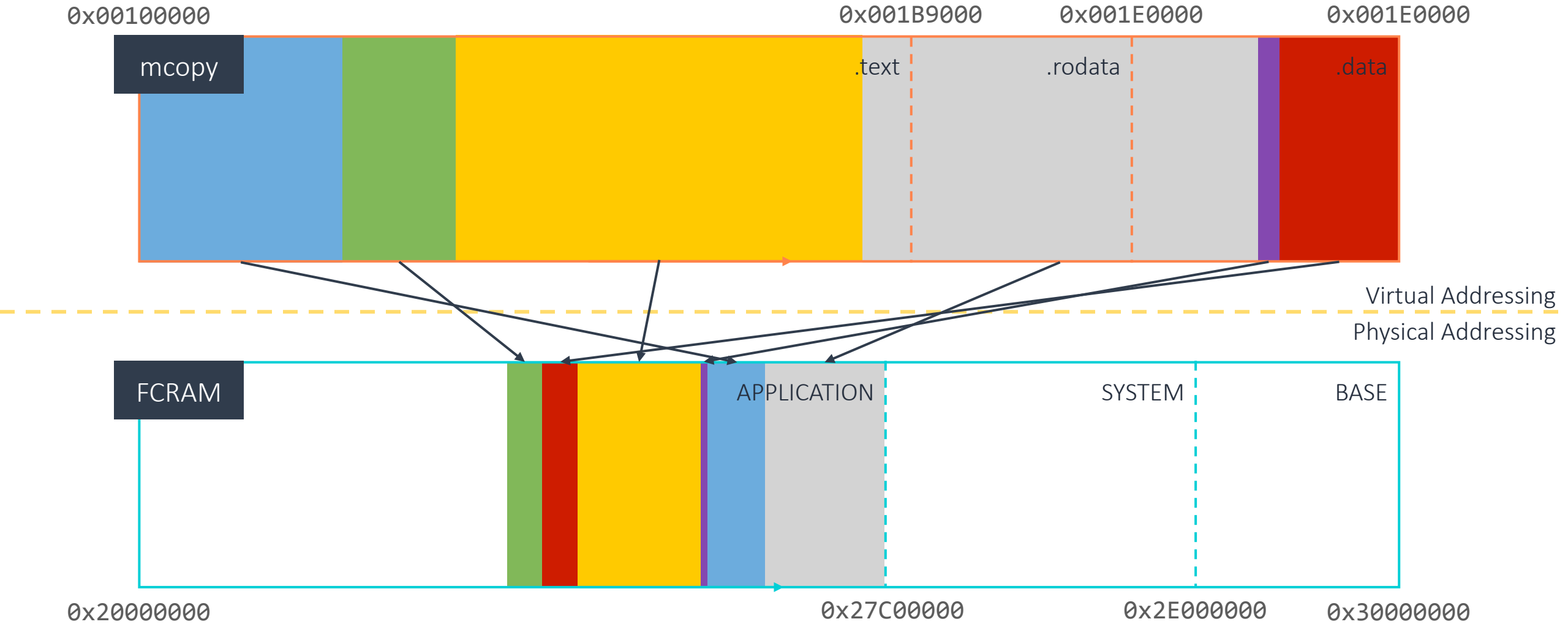
# GPU DMA




# GPU DMA: range reduction mitigation




# Using DMA to achieve code execution




# Nintendo’s mitigation: PASLR

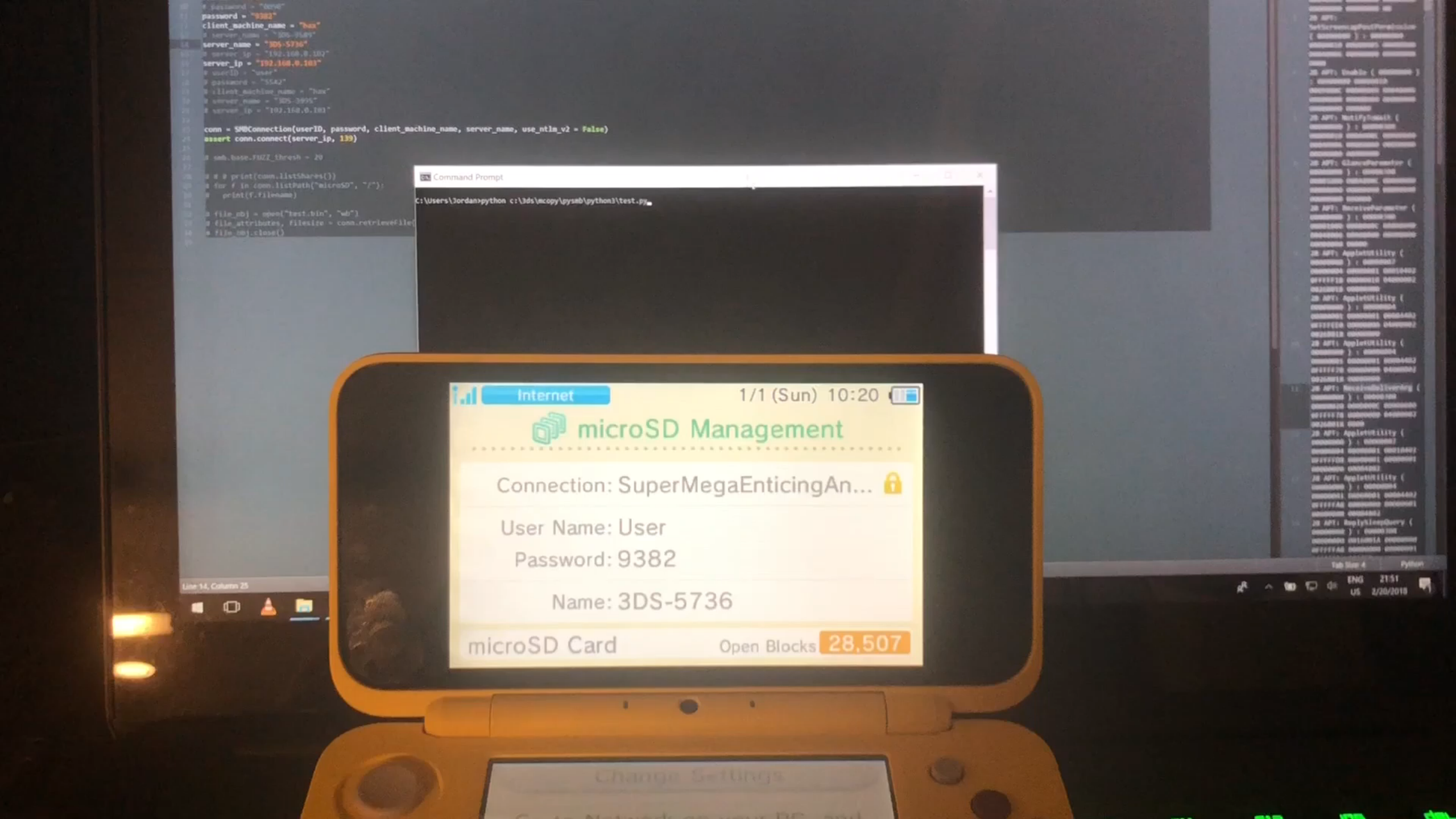
DMA PASLRed code data to CPU-readable location

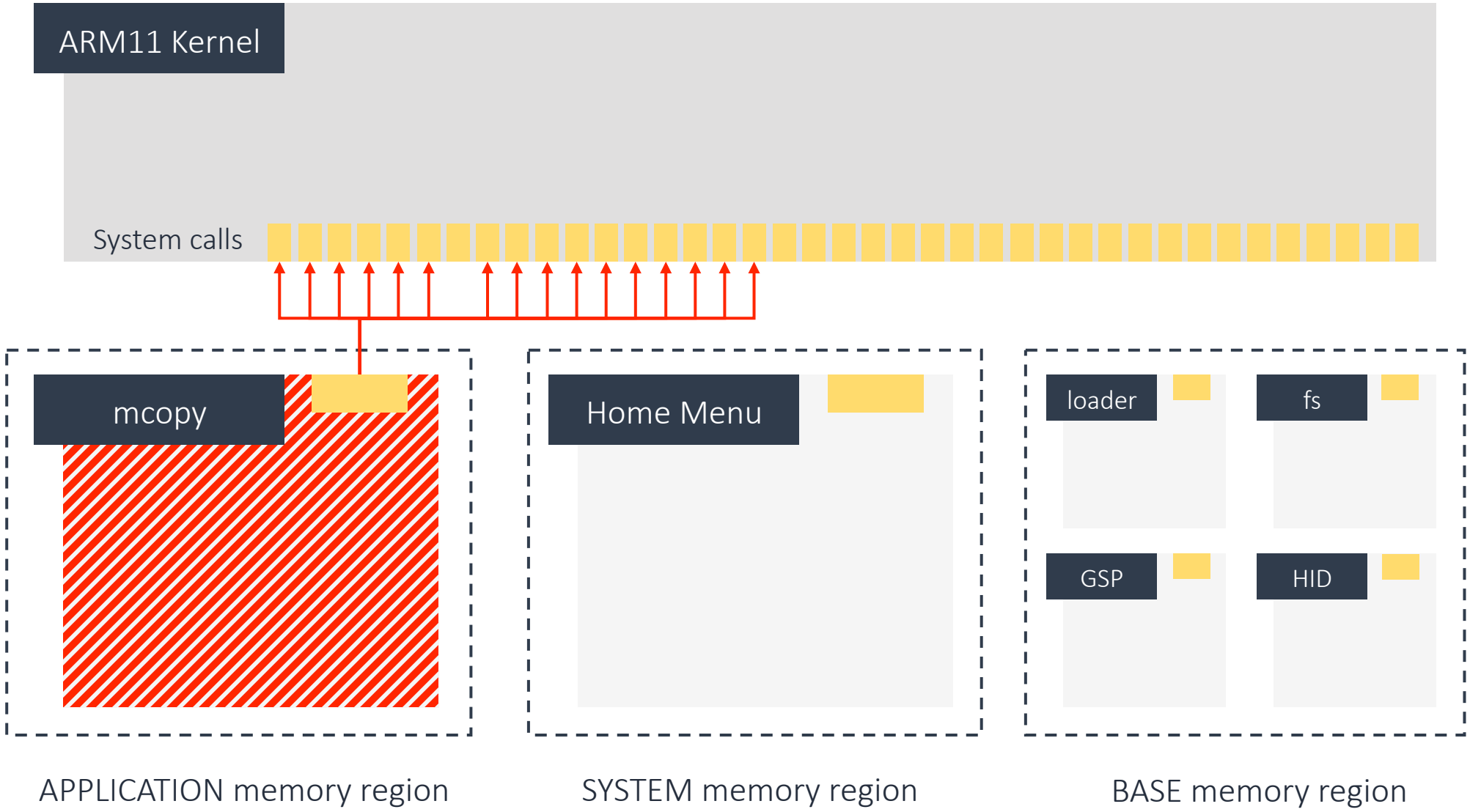
```
for(u32* ptr = MCOPY_RAND_COPY_BASE;
    *ptr != magic_value;
    ptr += MCOPY_SCANLOOP_STRIDE/4);
```

DMA code to known VA and jump to it

```
rop:
  gspwn MCOPY_RANDCODEBIN_COPY_BASE, MCOPY_RANDCODEBIN_BASE, MCOPY_CODEBIN_SIZE

  str_val MCOPY_SCANLOOP_CURPTR, MCOPY_RANDCODEBIN_COPY_BASE - MCOPY_SCANLOOP_STRIDE

  scan_loop:
    ldr_add_r0 MCOPY_SCANLOOP_CURPTR, MCOPY_SCANLOOP_STRIDE
    str_r0 MCOPY_SCANLOOP_CURPTR

    cmp_derefptr_r0addr MCOPY_SCANLOOP_MAGICVAL, scan_loop, scan_loop_pivot_after

    str_r0 scan_loop_pivot + 4

    scan_loop_pivot:
    jump_sp 0xDEADBABE
    scan_loop_pivot_after:

  memcpy MCOPY_RANDCODEBIN_COPY_BASE, initial_code, initial_code_end - initial_code

  flush_dcache MCOPY_RANDCODEBIN_COPY_BASE, 0x00100000

  gspwn_dstderefadd (MCOPY_RANDCODEBIN_BASE) - (MCOPY_RANDCODEBIN_COPY_BASE),
                    MCOPY_SCANLOOP_CURPTR, MCOPY_RANDCODEBIN_COPY_BASE, 0x800, 0

  .word MCOPY_SCANLOOP_TARGETCODE

.align 0x4
  initial_code:
    .incbin "../build/mhax_code.bin"
  initial_code_end:
```

# Bypassing PASLR in ROP


password = "9382"
client_machine_name = "hax"
server_name = "3DS-5736"
server_ip = "192.168.0.103"
conn = SMBConnection(userID, password, client_machine_name, server_name, use_ntlm_v2 = False)
assert conn.connect(server_ip, 139)
Command Prompt
C:\Users\Jordan>python c:\3ds\mcopy\pysmb\python3\test.py
Internet
1/1 (Sun) 10:20
microSD Management
Connection: SuperMegaEnticingAn...
User Name: User
Password: 9382
Name: 3DS-5736
microSD Card
Open Blocks 28,507
Change Settings




User-mode application compromised!

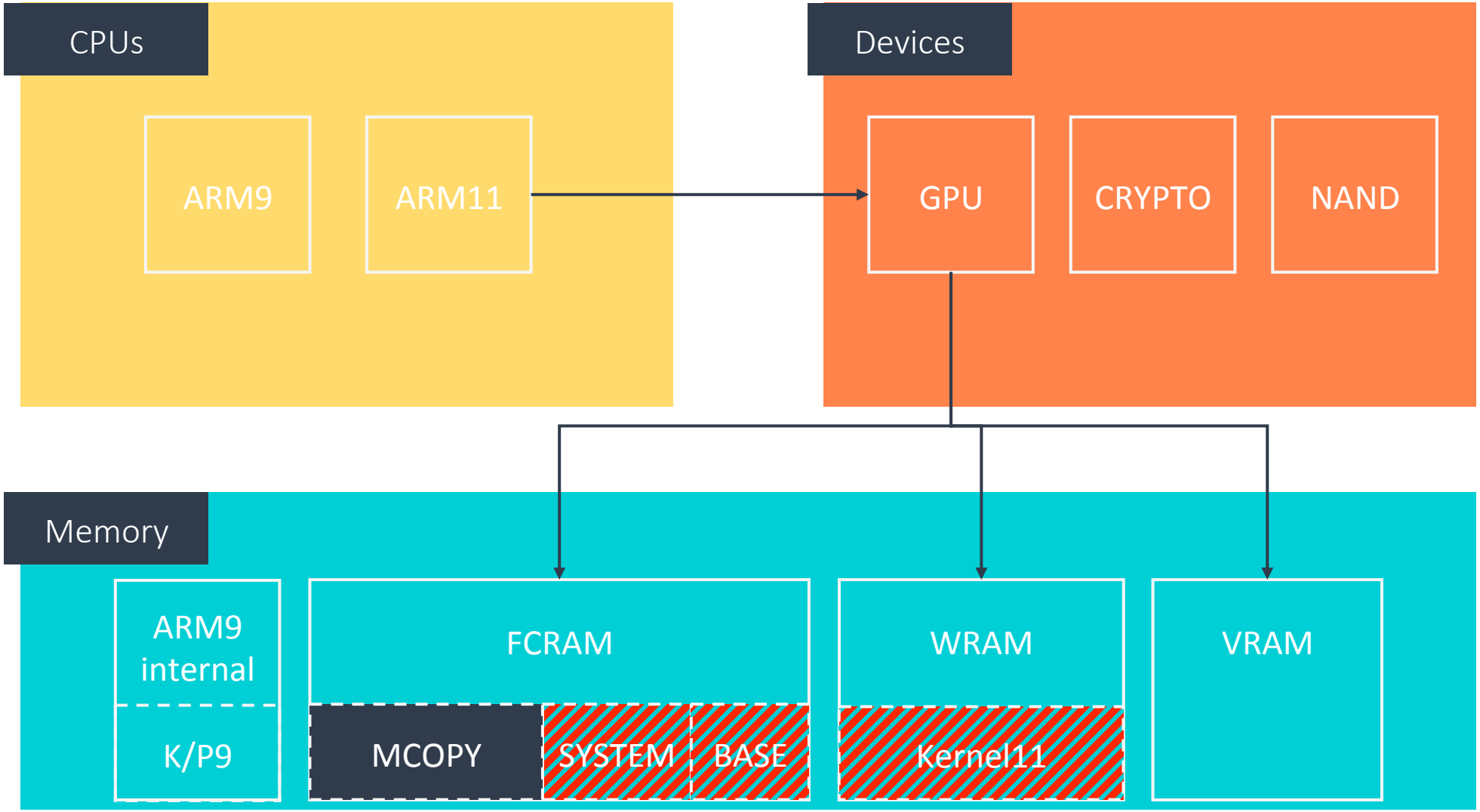
# Escalating privilege

We’re in! …now what?

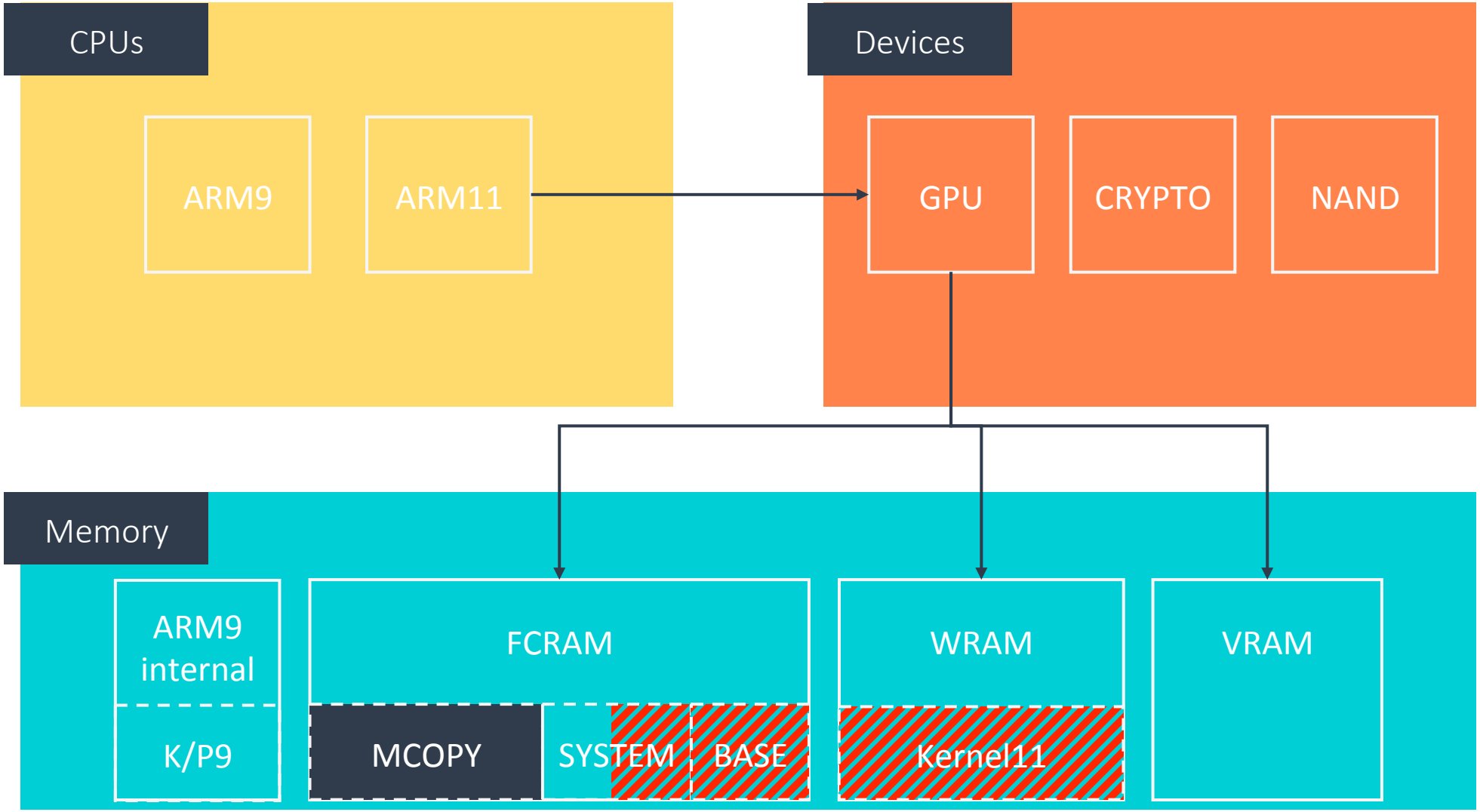


# Where do we stand?

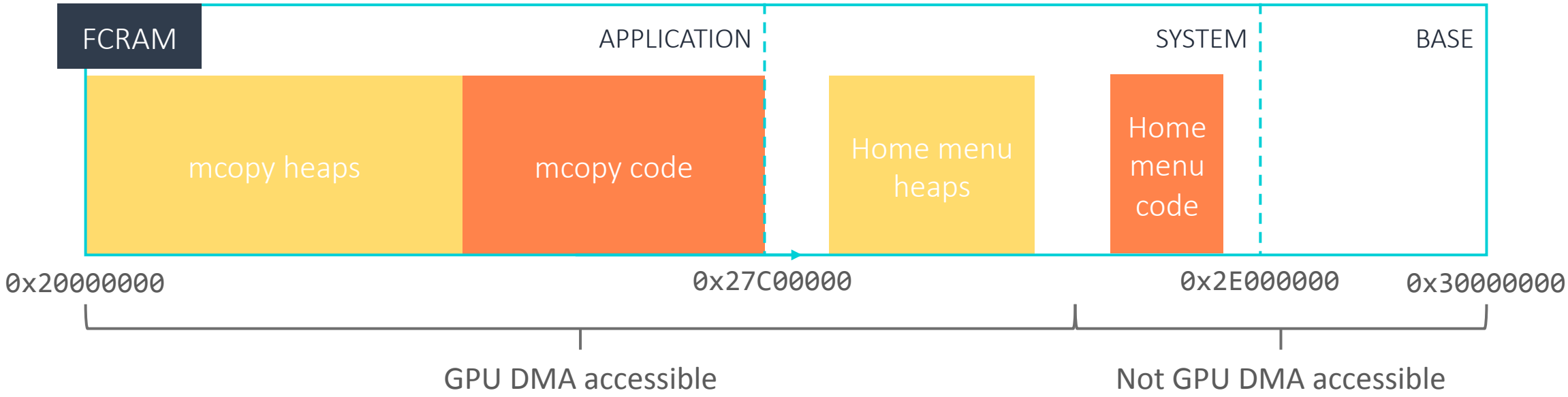
- mcopy is just an app
  - It only has access to basic system calls
  - It only has access to a few services
- Paths to exploitation
  - Directly attack the ARM9: difficult without access to more services
  - Attack the ARM11 kernel: definitely possible but easier with more system calls
  - Attack other user-mode processes




# GPU DMA: range reduction mitigation




GPU DMA range reduction: I lied




# FCRAM and GPU DMA

# Taking over home menu

- GPU DMA allows us to read/write home menu's heap

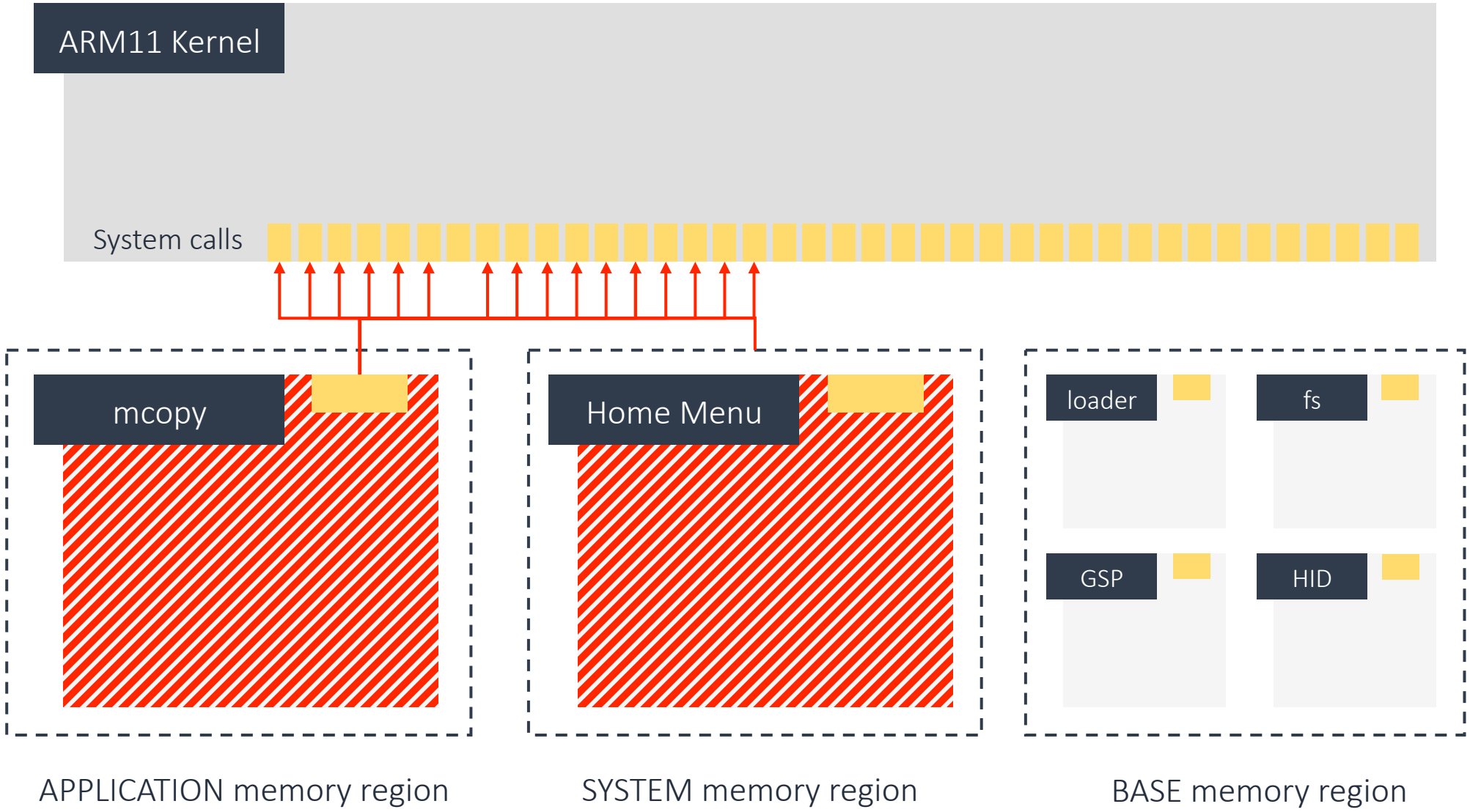
=> Find an interesting object, corrupt it and jump back to home menu

- Can't use GPU DMA to get code execution under home menu

=> Write a service in ROP that runs under home menu to give apps access to its privileges

# Side note: GPU DMA range mitigation

- Nintendo's idea
  - Different processes need different GPU DMA range
  - For example, apps never need to DMA to/from home menu
  - So why not restrict app/game's DMA more than home menu's?
- Implemented in 11.3.0-36, released on February 6th 2017
- Bypassed on New 3DS on February 10th
  - The problem: the DMA restriction doesn't cover home menu's  whole heap




Home menu compromised, giving access to more services

# Home menu’s privileges

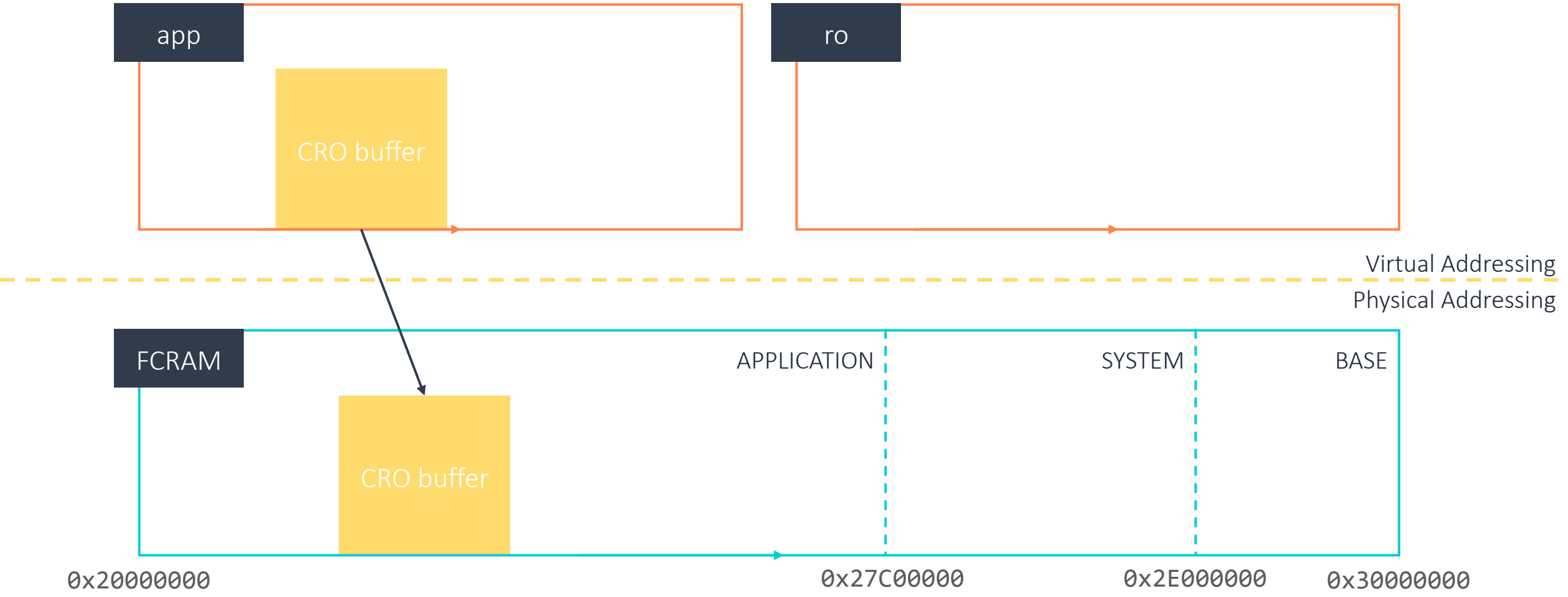
- Access to the ns:s service
  - NS: Nintendo Shell
  - Allows us to kill and spawn processes at will

⇒ We can access any service accessible from an app

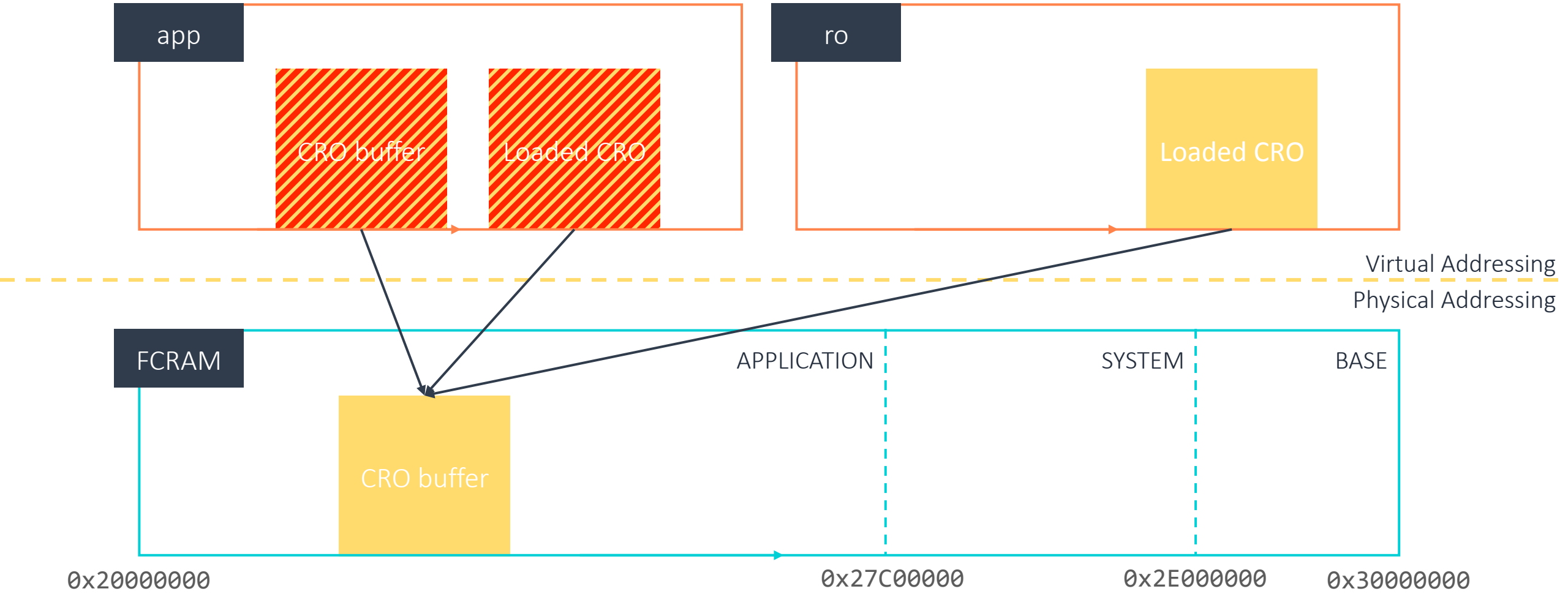
  - Use ns:s to spawn the app
  - Use GPU DMA to overwrite its code and take it over
  - Access the service from that app

# ldr:ro

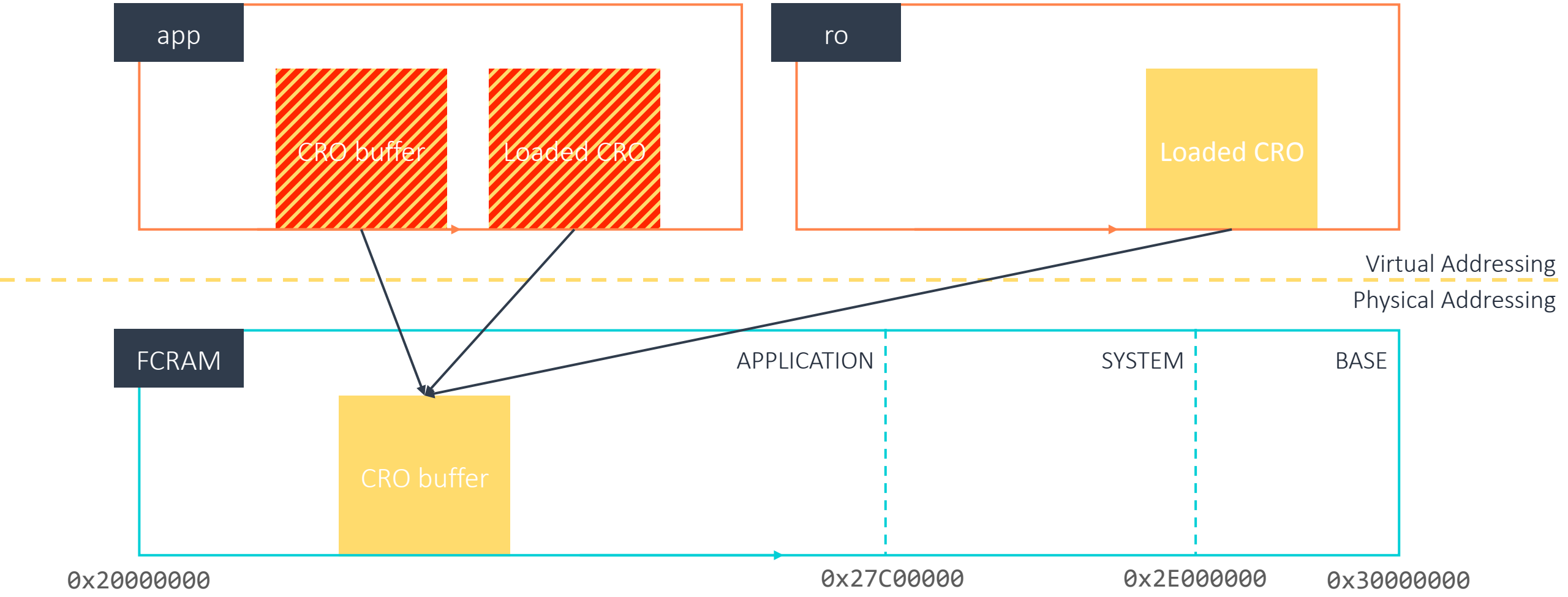
- Service provided by the “ro” process
- Handles loading dynamic libraries: CROs
  - Basically like DLLs for the 3DS
- Is the only process to have access to certain system calls
  - Most interesting one: svcControlProcessMemory
  - Lets you allocate/reprotect memory as RWX
  - Useful for homebrew among other things…



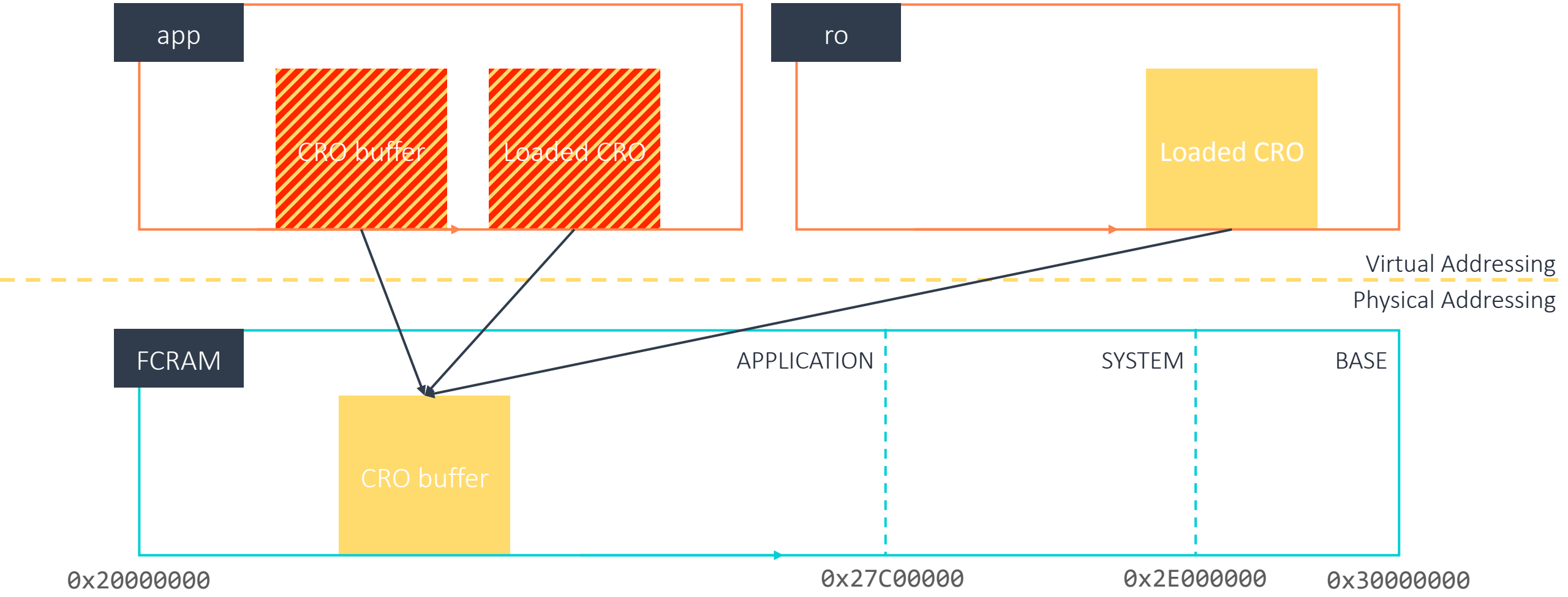

# ldr:ro: first, application loads CRO into an RW buffer



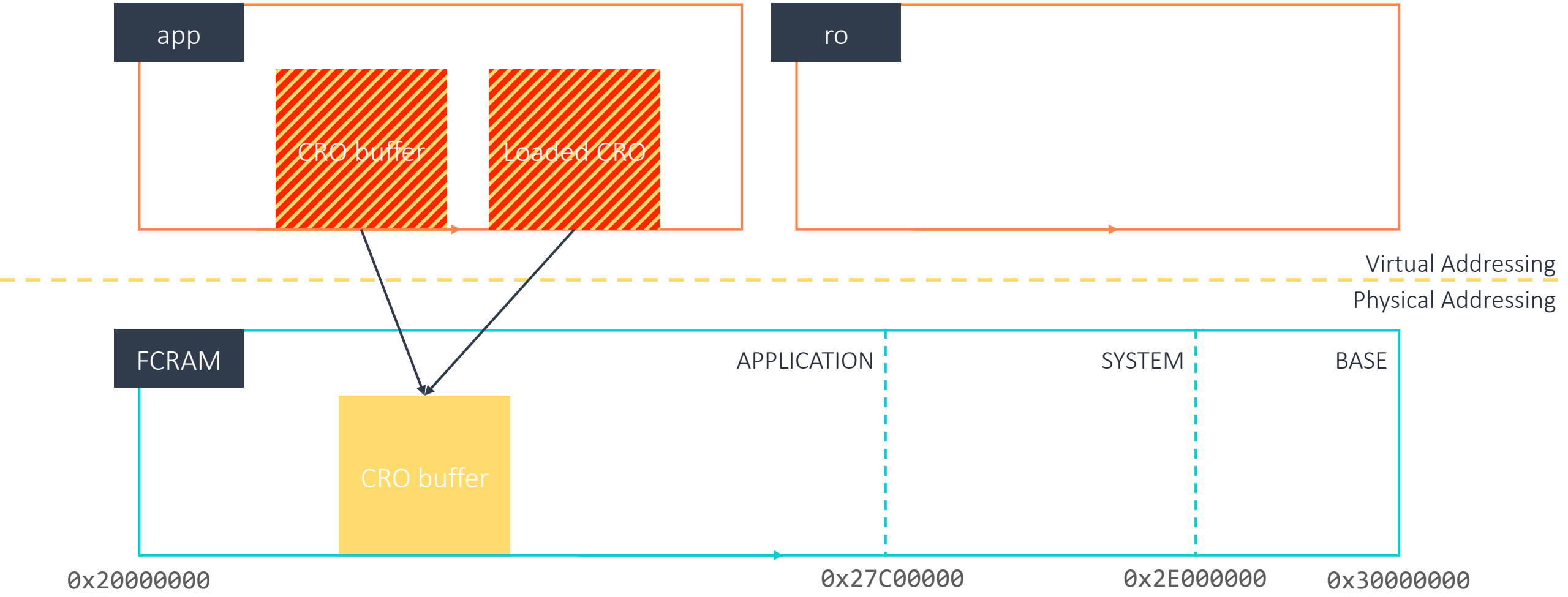

# ldr:ro: second, CRO is locked for the app and mapped to its load address



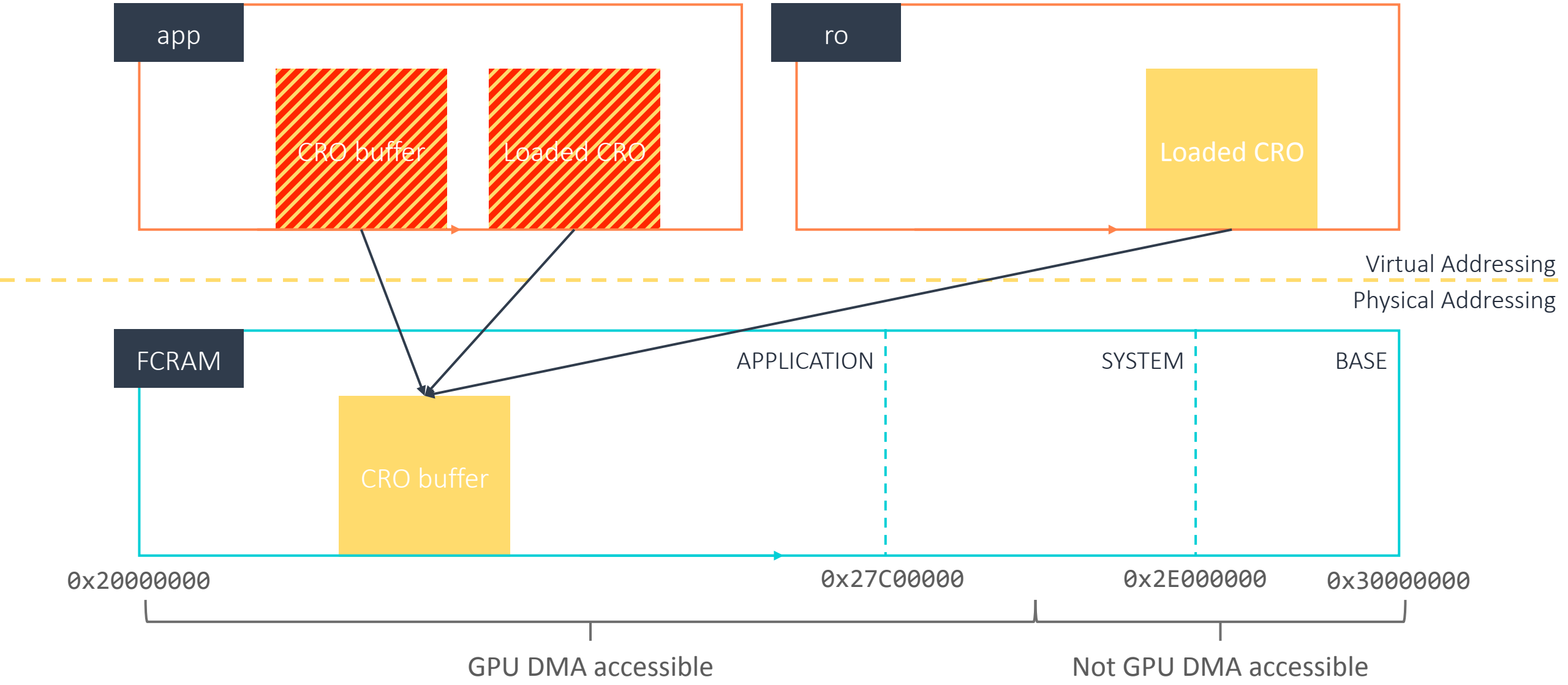

ldr:ro: third, ro creates a local view of the CRO in its own memory space




ldr:ro: fourth, ro performs processing on CRO (relocations, linking etc)



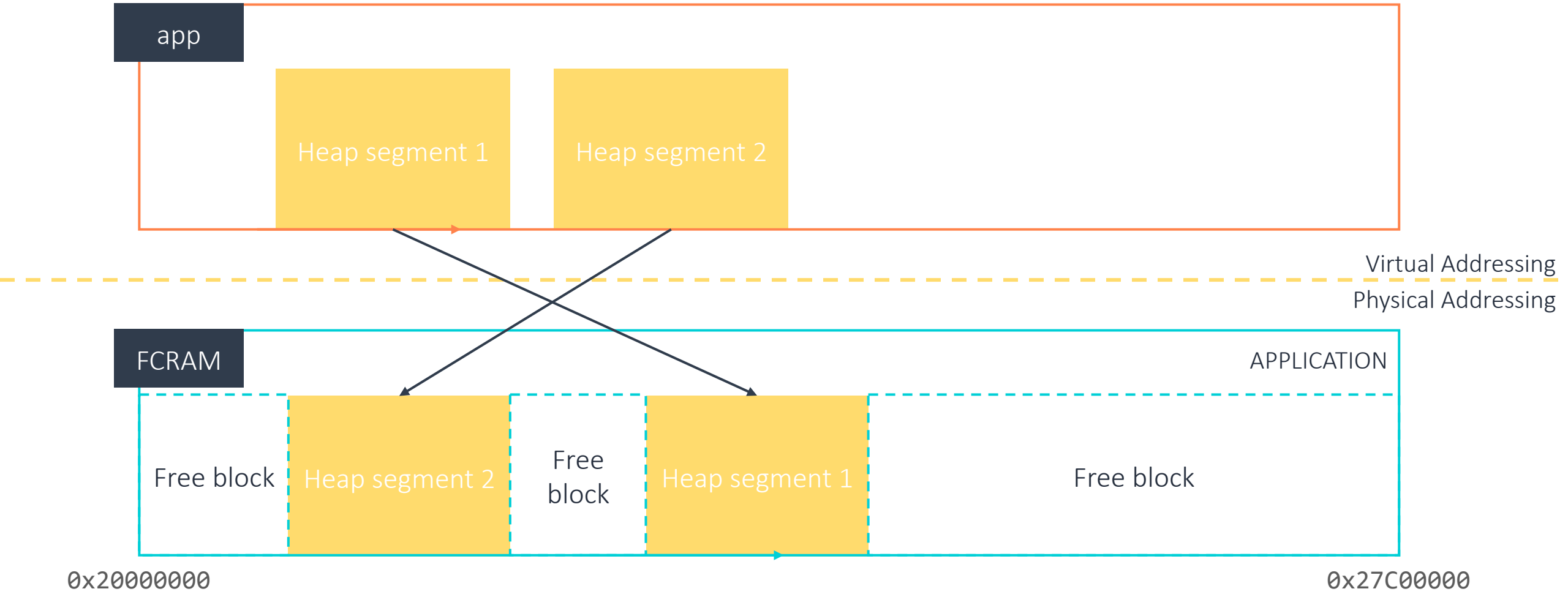

# ldr:ro: finally, ro unmaps the CRO and reprotects the app's loaded view



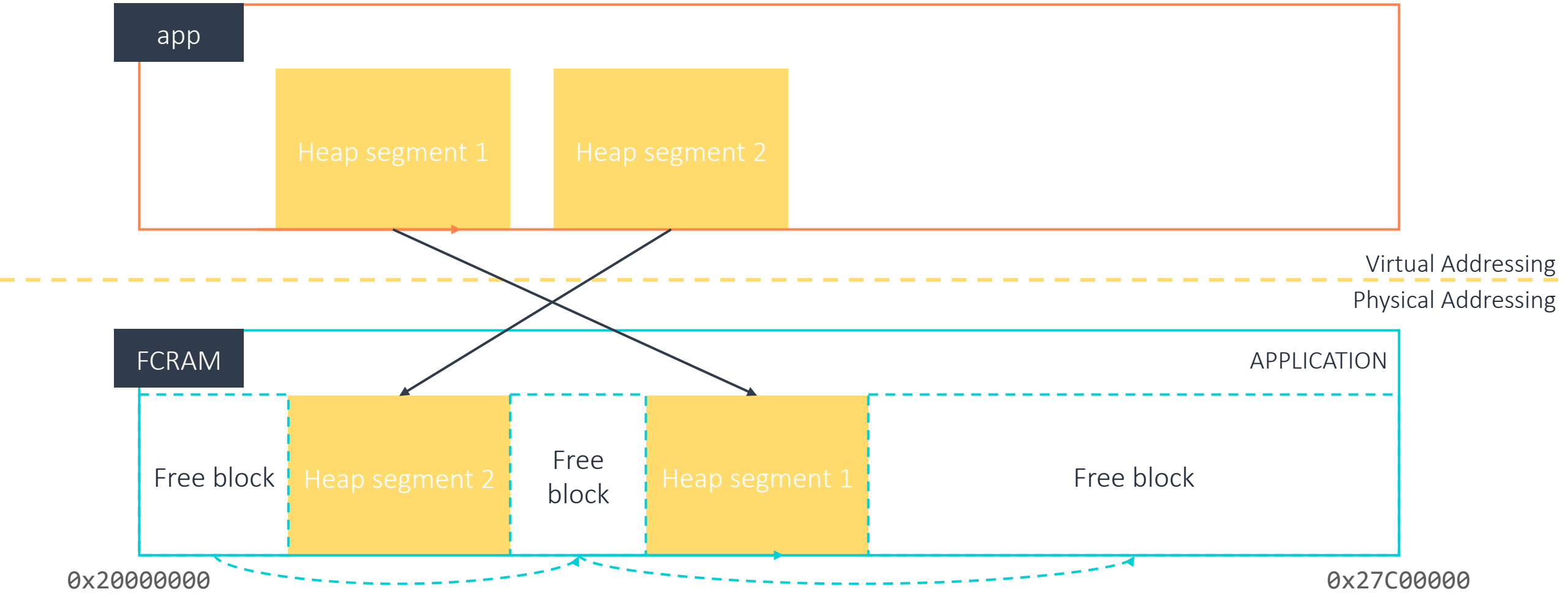

Key insight: app can’t modify CRO from CPU, but can with GPU

# CRO tampering with GPU

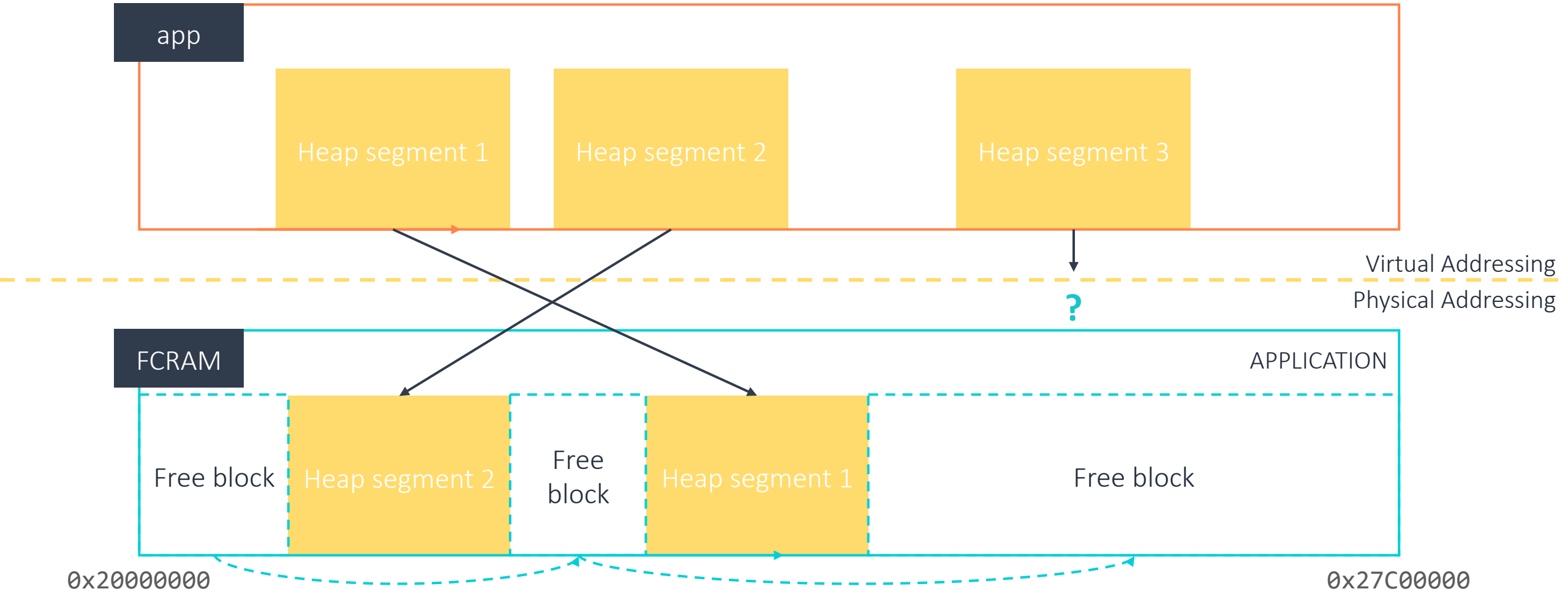
- Nintendo's CRO loader is written with this in mind
  - Lots of checks to prevent malformed CROs from compromising ro process
- However, Nintendo didn't account for modifying CRO *during* processing
  - Lots of possible race condition bugs!
- Using GPU DMA for time-critical memory modification is tricky, especially with cache in the middle
  - Kernel prevents us from double-mapping the CRO memory…
  - …in theory



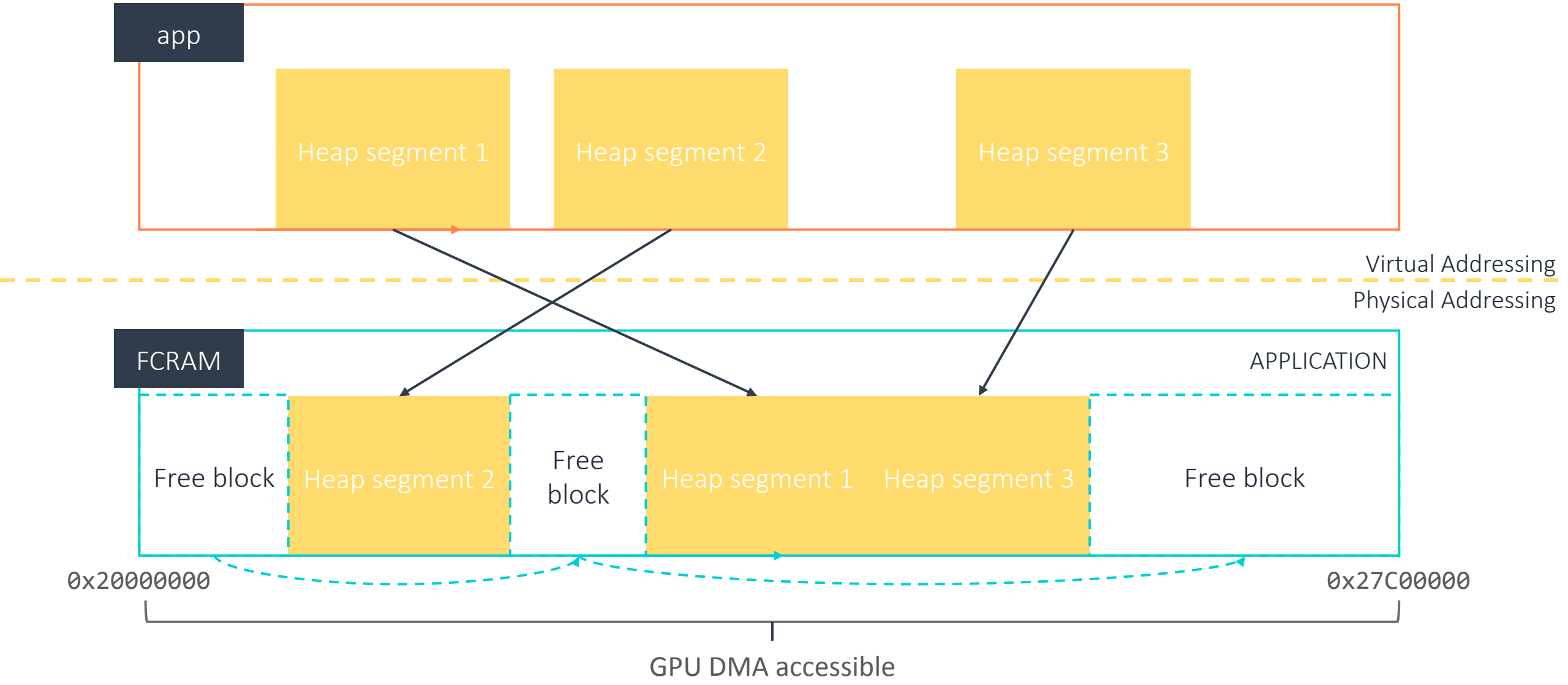

Kernel keeps physical heap metadata in free physical memory blocks




The metadata is essentially just a linked list



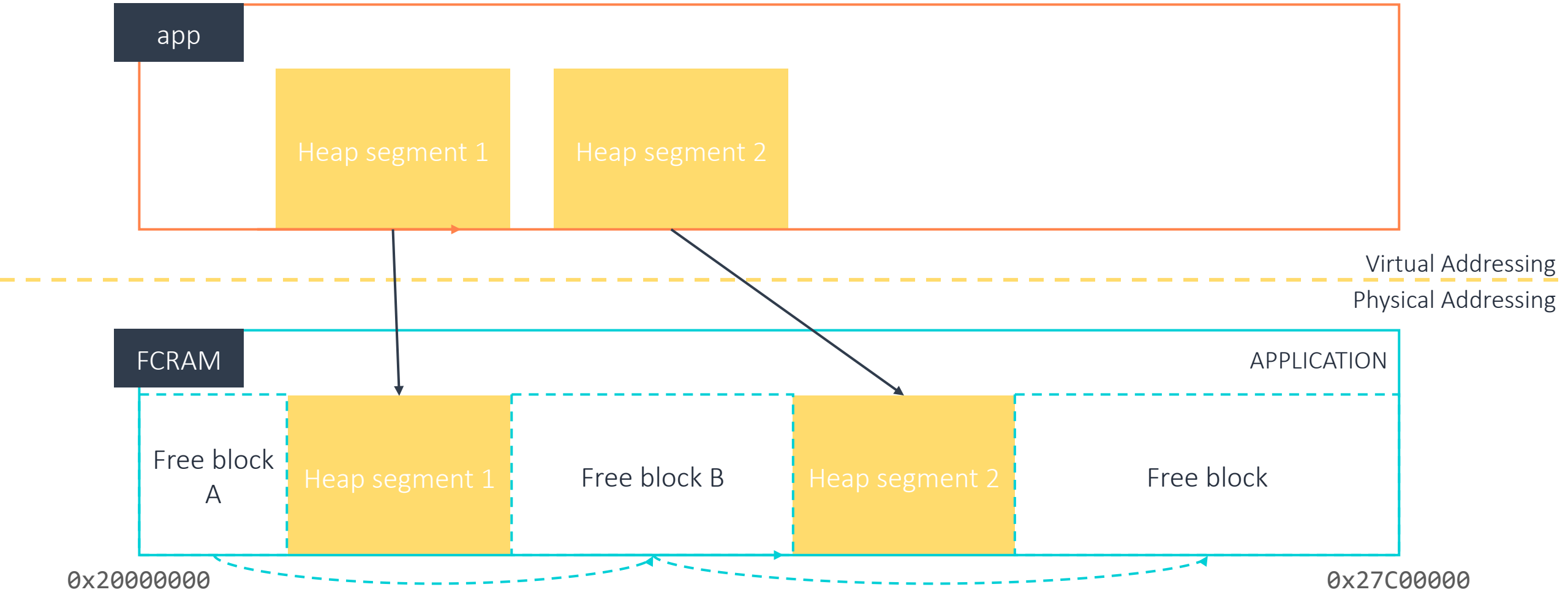

When allocating a new heap segment, the kernel just walks the list



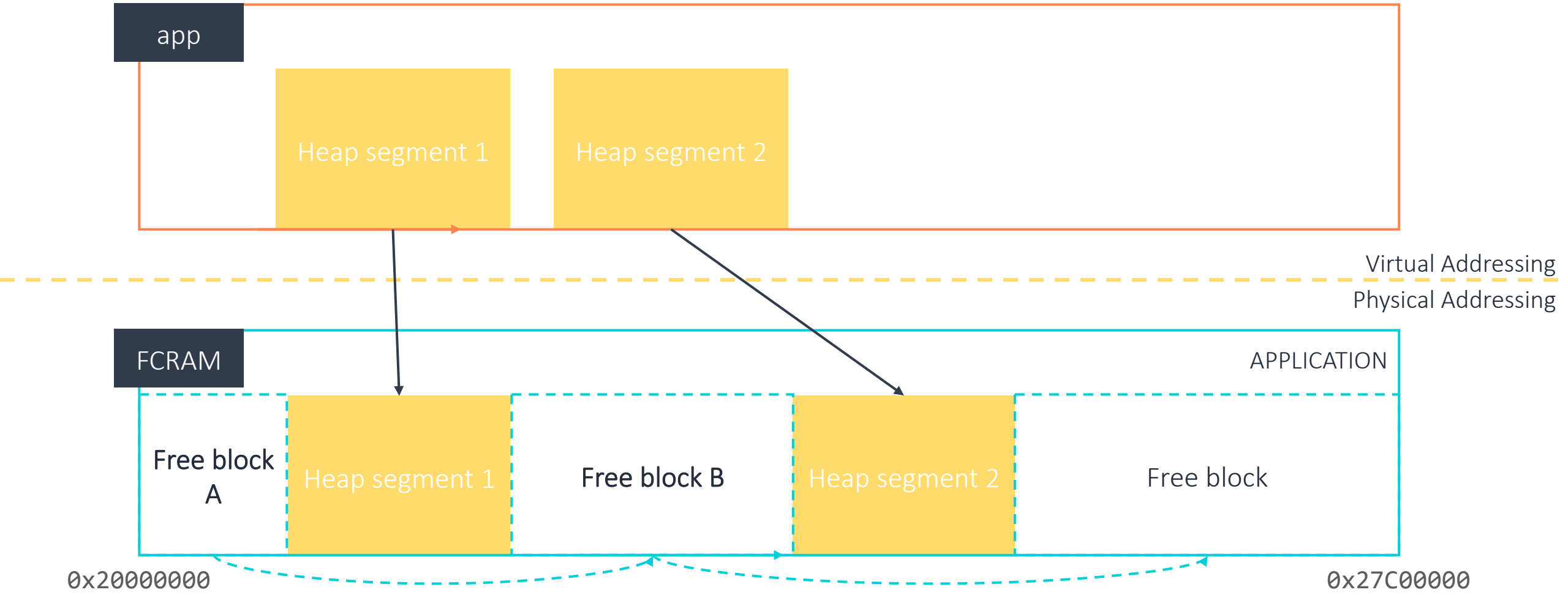

# Again: app can’t modify heap metadata from CPU, but can with GPU

# Heap metadata authentication

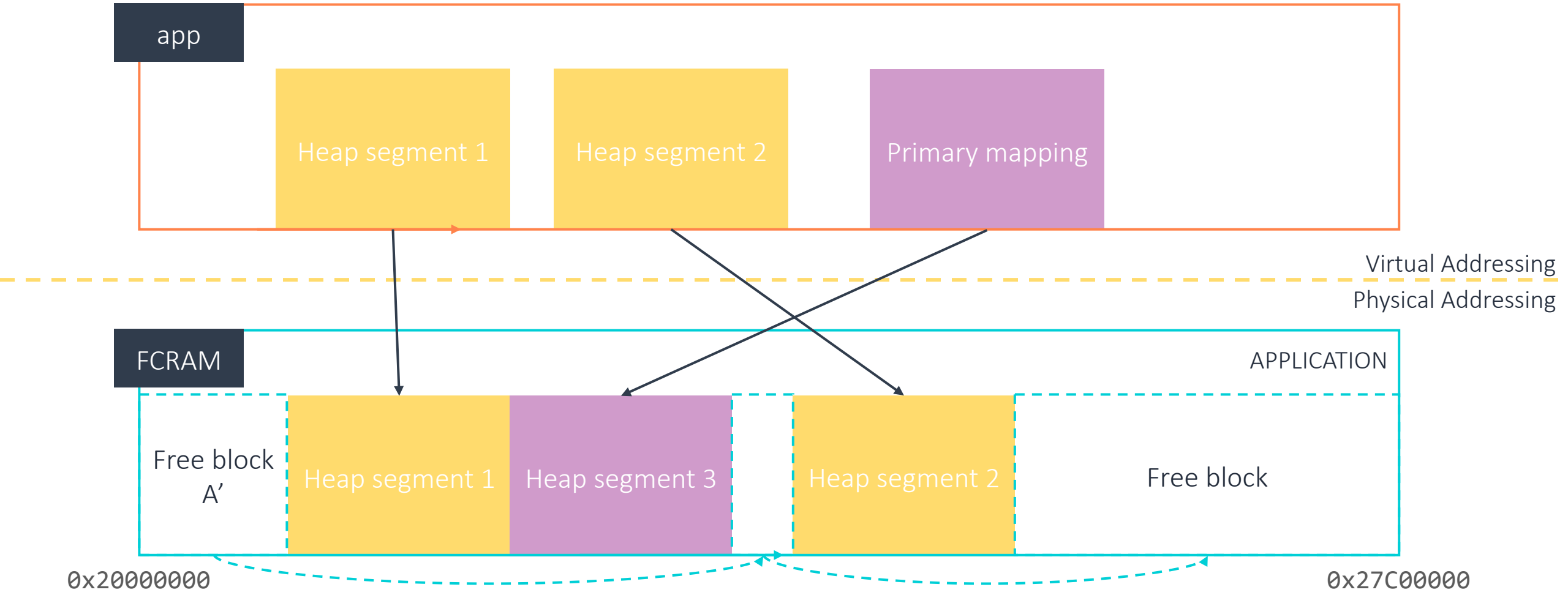
- Nintendo knows kernel-trusted DMA-able heap metadata is bad
- Introduced a MAC into the metadata with a key only known to kernel
- Prevents forgery of arbitrary heap metadata blocks…
- … but not replay attacks



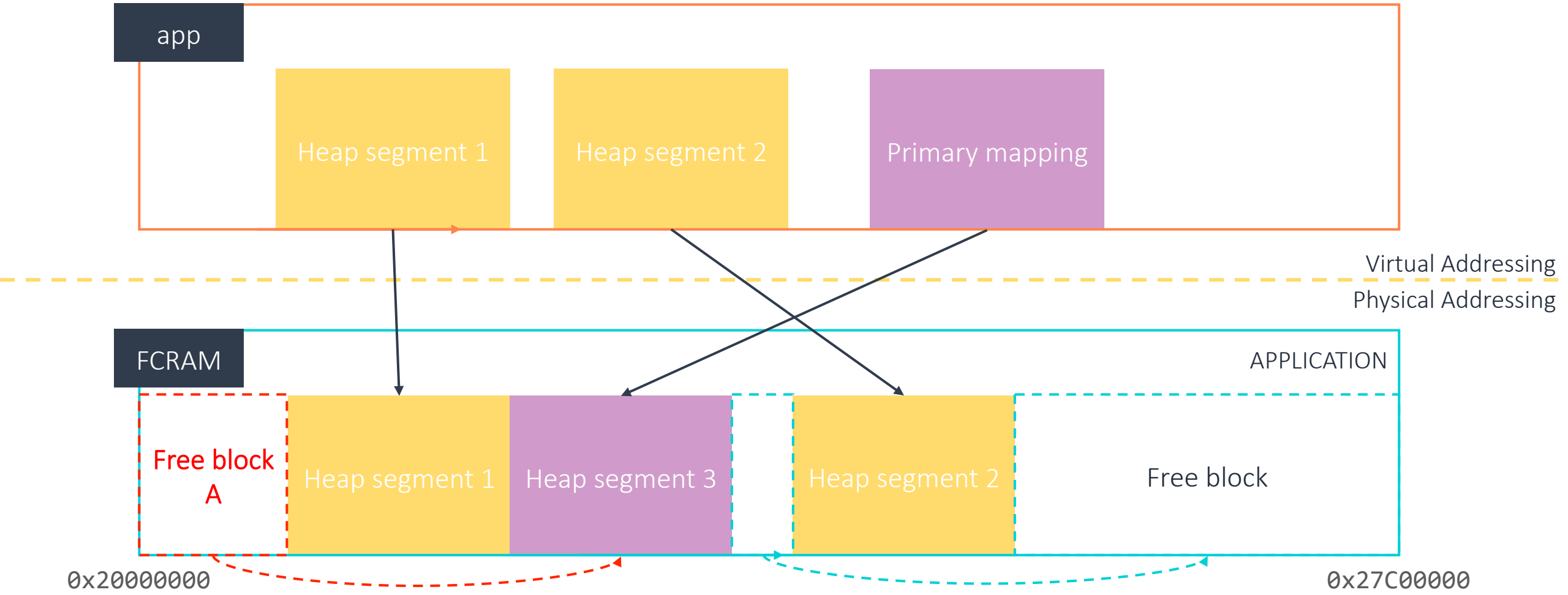

# Creating a double mapping: initial layout



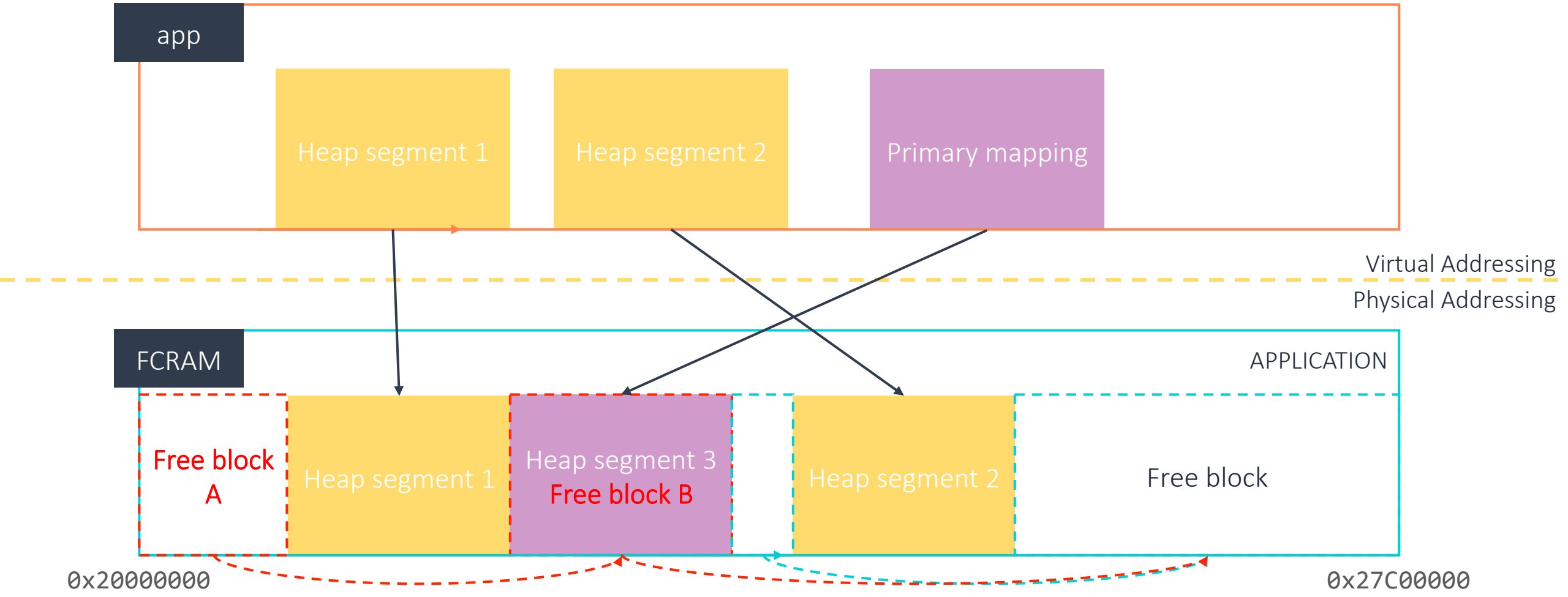

# Creating a double mapping: save free block A and B's data through DMA



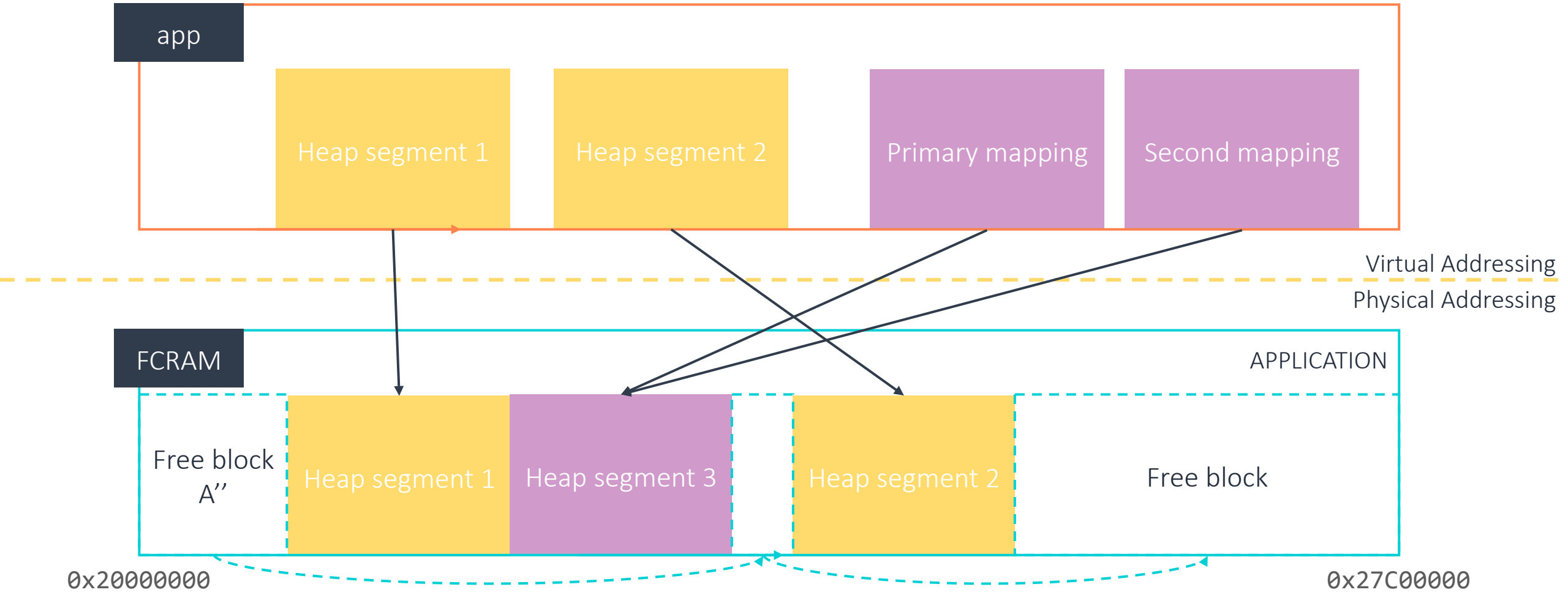

# Creating a double mapping: allocate segment to fit in B but not A




# Creating a double mapping: use DMA to replace A’ with A




# Creating a double mapping: write B's data to heap segment 3




# Creating a double mapping: allocate second mapping

Relocates offsets as pointers in the CRO buffer, after checking them

Attacker may have time to modify the CRO buffer

ro uses pointers loaded from the CRO buffer without double checking

```
...

u32 segment_table_offset = *(u32*)&cro_buf[0xC8];
if ( segment_table_offset )
{
  void* segment_table_ptr = &cro_buf[segment_table_offset];

  if ( is_in_cro_bounds(segment_table_ptr) )
  {
    *(u32*)&cro_buf[0xC8] = (u32)segment_table_ptr;
  } else goto fail;
}

...

u32 num_segments = *(u32*)&cro_buf[0xCC];

for(int i = 0; i < num_segments; i++)
{
  cro_segment_s* segment_table = *(cro_segment_s**)&cro_buf[0xC8];
  cro_segment_s* cur_segment = &segment_table[i];

  ...
}

if(!everything_ok) throw_error(0xD9012C19);
```

# ldr:ro race condition

Attacker-controlled value (race condition)

Attacker-controlled value (parameter)

**A.** Can write an arbitrary value to X if:
- `*(u8*)(X + 8) == 0x02`
- `*(u32*)(X + 4) != 0`

**B.** Can write an arbitrary value to X if:
- `*(u8*)(X + 8) == 0x03`
- `*(u32*)(X + 4) != 0`

**C.** Can add a semi-arbitrary value at X if:
- `*(u8*)(X + 8) not in [0x03, 0x02]`
- `*(u32*)X != 0`
- Added value must be page-aligned

```
cro_segment_s* segment_table = *(cro_segment_s**)&cro_buf[0xC8];
cro_segment_s* cur_segment = &segment_table[i];

switch(cur_segment->id)
{
  case 2: // CRO_SEGMENT_DATA
    if ( !cur_segment->size ) continue;
    if ( cur_segment->size > data_size ) throw_error(0xE0E12C1F);
    cur_segment->offset = data_adr;
    break;
  case 3: // CRO_SEGMENT_BSS
    if ( !cur_segment->size ) continue;
    if ( cur_segment->size > bss_size ) throw_error(0xE0E12C1F);
    cur_segment->offset = bss_adr;
    break;
  default:
    if(everything_ok && cur_segment->offset)
    {
      u32 cur_segment_target = cro_buf + cur_segment->offset;
      cur_segment->offset = cur_segment_target;
      if(cro_buf > cur_segment_target
        || cro_buf_end < cur_segment_target) everything_ok = false;
    }
}
if(!everything_ok) throw_error(0xD9012C19);
```

# ldr:ro race condition

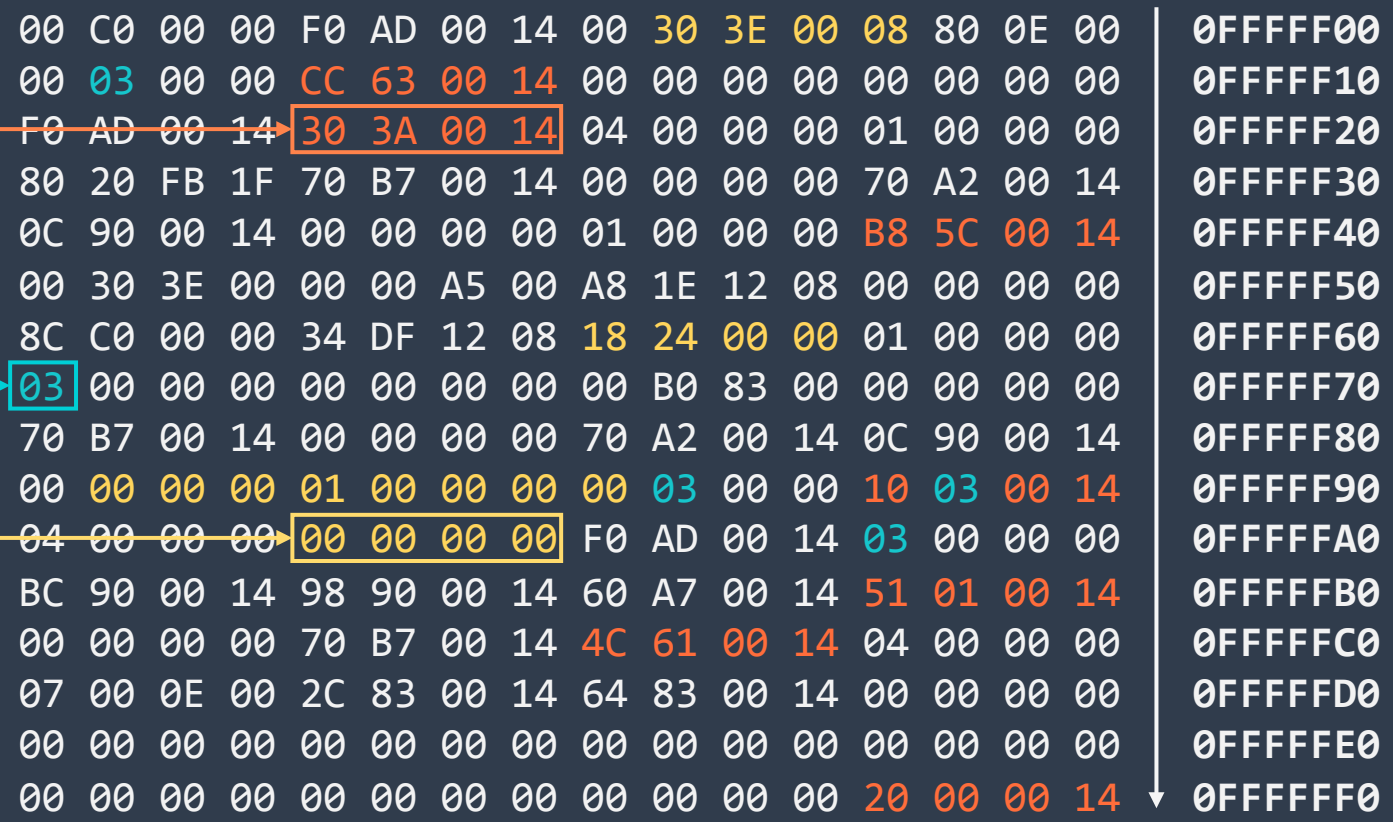
Return addresses: what we'd like to corrupt

0x03 bytes allowing arbitrary value writes

Memory which we can arbitrarily overwrite

…no overlap…

## ro call stack data




# Getting ROP in ro: arbitrary value write

**C.** Can add a semi-arbitrary value at X if:

- `*(u8*)(X + 8) not in [0x03, 0x02]?`
  - `0x8121EAE != 0x03 and 0x02` ✓
- `*(u32*)X != 0?`
  - `0x3E3000 != 0` ✓
- Added value must be page-aligned?
  - `0xC50000 is page-aligned` ✓

## ro call stack data

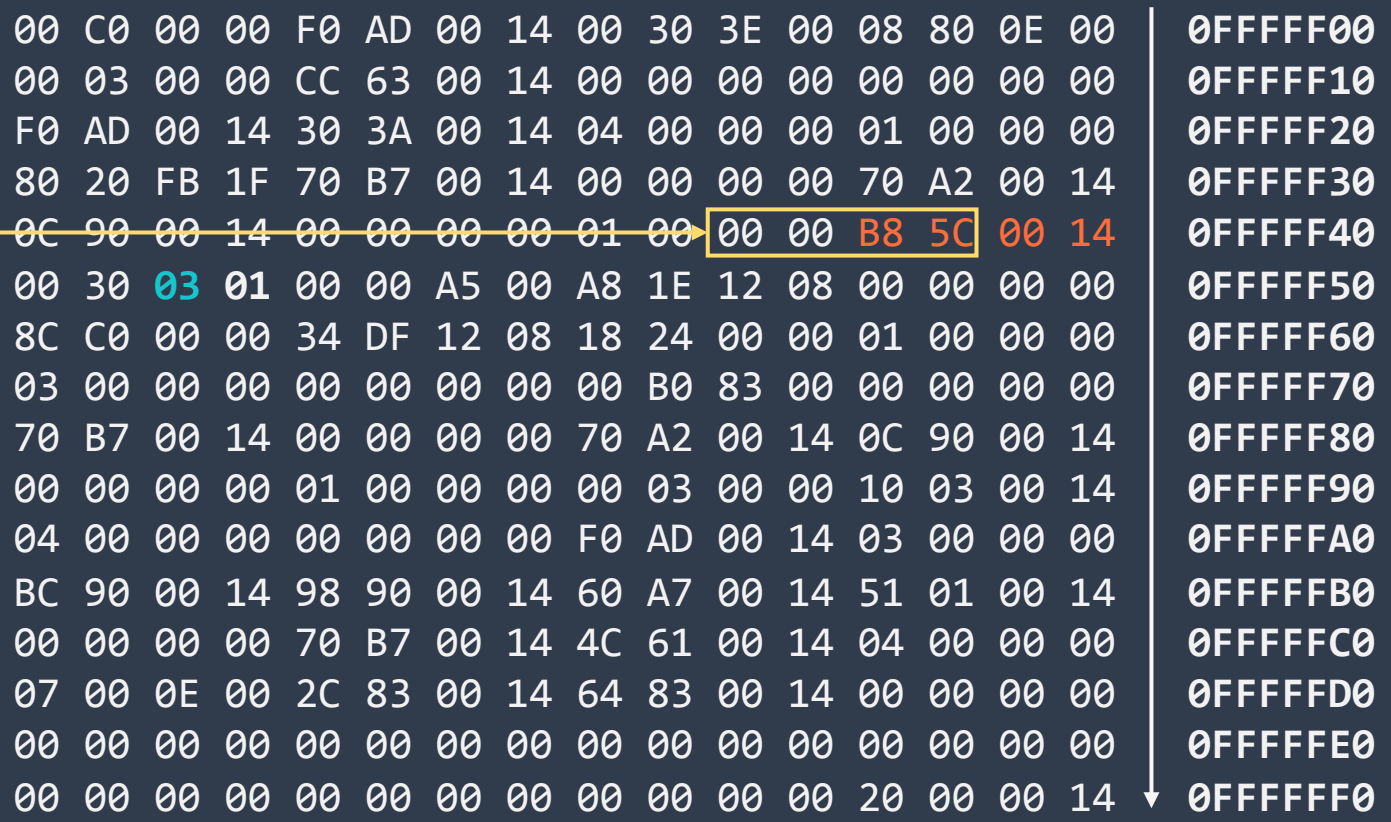
```
00 C0 00 00 F0 AD 00 14 00 30 3E 00 08 80 0E 00 | 0FFFFF00
00 03 00 00 CC 63 00 14 00 00 00 00 00 00 00 00 | 0FFFFF10
F0 AD 00 14 30 3A 00 14 04 00 00 00 01 00 00 00 | 0FFFFF20
80 20 FB 1F 70 B7 00 14 00 00 00 00 70 A2 00 14 | 0FFFFF30
0C 90 00 14 00 00 00 00 01 00 00 00 B8 5C 00 14 | 0FFFFF40
00 30 3E 00 00 00 A5 00 A8 1E 12 08 00 00 00 00 | 0FFFFF50
8C C0 00 00 34 DF 12 08 18 24 00 00 01 00 00 00 | 0FFFFF60
03 00 00 00 00 00 00 00 00 B0 83 00 00 00 00 00 | 0FFFFF70
70 B7 00 14 00 00 00 00 70 A2 00 14 0C 90 00 14 | 0FFFFF80
00 00 00 00 01 00 00 00 00 03 00 00 10 03 00 14 | 0FFFFF90
04 00 00 00 00 00 00 00 F0 AD 00 14 03 00 00 00 | 0FFFFFA0
BC 90 00 14 98 90 00 14 60 A7 00 14 51 01 00 14 | 0FFFFFB0
00 00 00 00 70 B7 00 14 4C 61 00 14 04 00 00 00 | 0FFFFFC0
07 00 0E 00 2C 83 00 14 64 83 00 14 00 00 00 00 | 0FFFFFD0
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | 0FFFFFE0
00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 14 | 0FFFFFF0
```

# Getting ROP in ro: combined primitives

**B.** Can write an arbitrary value to x if:

- `*(u8*)(X + 8) == 0x03`
  - `0x03 == 0x03` ✓
- `*(u32*)(X + 4) != 0`
  - `0x30001400 != 0` ✓

## ro call stack data

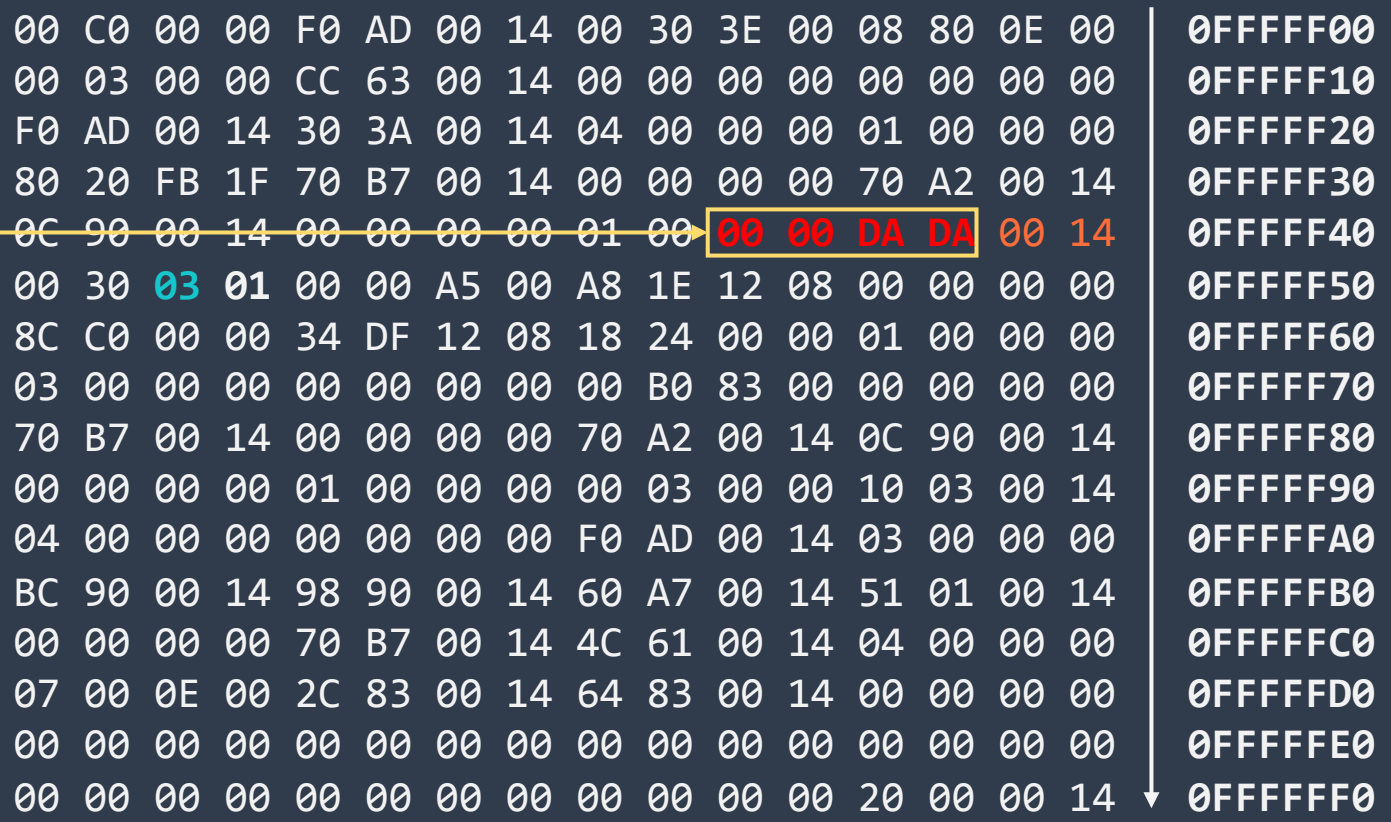



# Getting ROP in ro: combined primitives

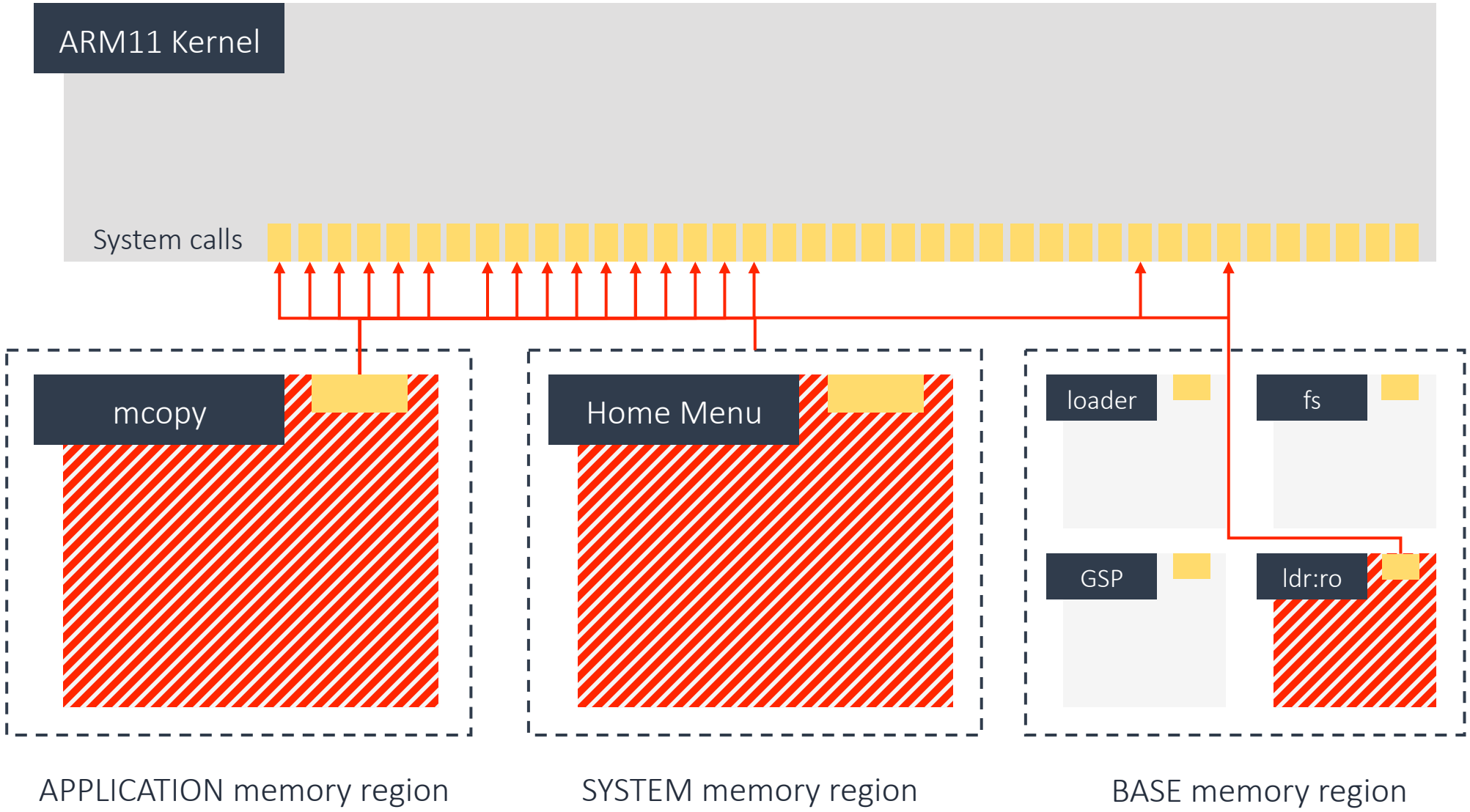
**B.** Can write an arbitrary value to x if:

- `*(u8*)(X + 8) == 0x03`
  - `0x03 == 0x03` ✓
- `*(u32*)(X + 4) != 0`
  - `0x30001400 != 0` ✓

## ro call stack data




# Getting ROP in ro: combined primitives




ldr:ro compromised, giving access to exotic system calls



# Taking over the ARM11

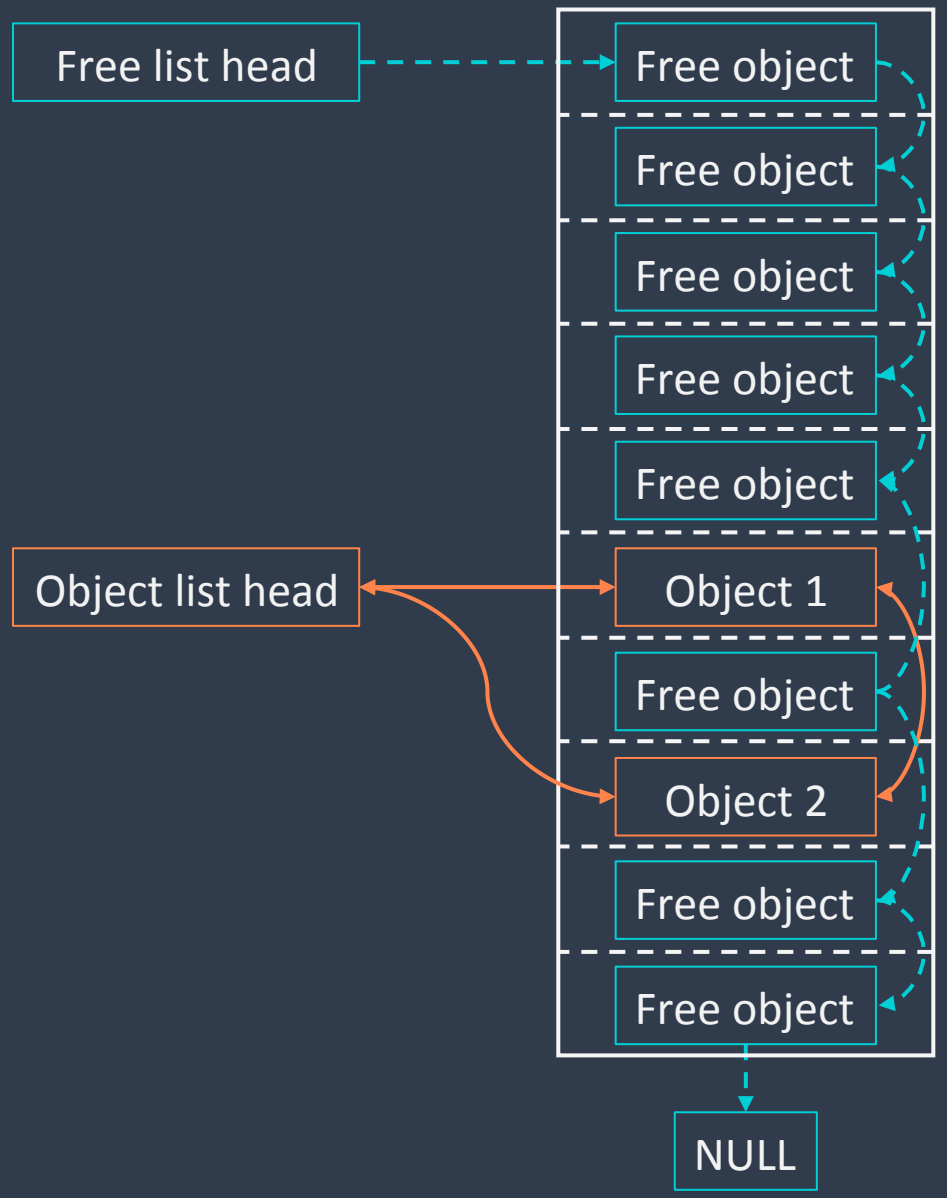
User mode is for losers

# svcControlProcessMemory

- Privileged system call
  - Only the ro system module has access to it
- Basically svcControlMemory but cross-process
  - Can allocate, reprotect and remap memory
  - Requires a handle to the target process
- Less constrained than svcControlMemory
  - Can allocate and protect memory as RWX!
  - Can map the NULL page…
- By design mitigation bypass: allows us to attack kernel NULL derefs
  - What's an easy NULL-deref target? Allocation code

# How are kernel objects allocated?

- Memory “slab” subdivided into same-size objects
- Objects are part of a free list when not in use
- Allocation = pop from list
- Freeing = push to list
- 3DS: one slab per object type
  - *Finite* number of each type of objects…
  - What happens if we run out?




Slab heap

Reads the head of the free list (with synchronization)

Pops the head of the free list (with synchronization)

No further checks or exception throws – `alloc_kobj` returns NULL when list is empty

```
KLinkedListNode* alloc_kobj(KLinkedListNode* freelist_head)
{
  KLinkedListNode* ret;

  do
  {
    ret = __ldrex(freelist_head);
  } while(__strex(ret ? ret->next : NULL, freelist_head));

  return ret;
}
```

# Slab heap allocation code



```
0xFFF0701C:
  v11 = alloc_kobj(freelist_1);
  if ( v11 )
  {
    ...
  }else{
    throw_error(0xC8601808);
  }
```

---



```
0xFFF086AC:
  v13 = alloc_kobj(freelist_2);
  if ( v13 )
  {
    ...
  }else{
    throw_error(0xD8601402);
  }
```

---



```
0xFFF22794:
  KLinkedListNode* node = alloc_kobj(freelist_listnodes);
  if ( node )
  {
    node->next = 0;
    node->prev = 0;
    node->element = 0;
  }
  node->element = ...;
```

# alloc_kobj uses

# svcWaitSynchronizationN

- Unprivileged system call
- Takes in a list of kernel objects and waits on them
  - Kernel objects to wait on: port, mutex, semaphore, event, thread…
  - Calling Thread goes to sleep until one of the objects signals
- Can wait on up to 256 objects at a time
- How does it keep track of objects it's waiting on? 🤔

# How to trigger a NULL deref

1. Create thread
2. Have thread wait on 256 objects
3. Have we dereferenced NULL yet?

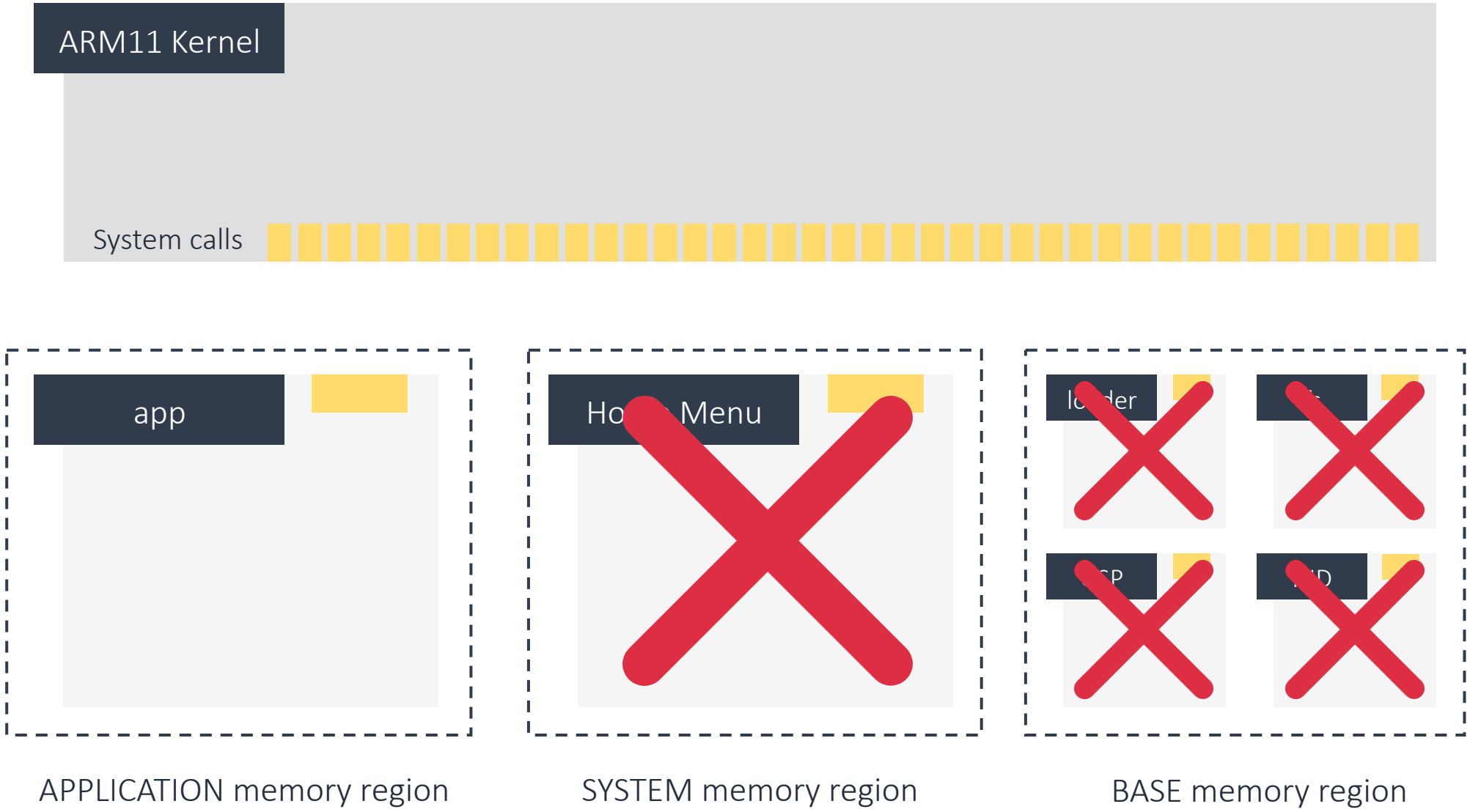
   No? Go to 1.

   Yes? We're done.

```
svcWaitSynchronizationN:
...

for ( int i = 0; i < num_kobjects; i++ )
{
  KObject* obj = kobjects[i];
  KLinkedListNode* node = alloc_kobj(freelist_listnodes);

  if ( node )
  {
    node->next = 0;
    node->prev = 0;
    node->element = 0;
  }

  node->element = obj;
  thread->wait_object_list->insert(node);
}

...
```

svcWaitSynchronizationN

Note: we can’t actually kill every single process out there but we can kill like 90% and that’s enough




**Problem 1 solution**: use ns:s service to kill every process we can except our own

# Problem 2 solution

- We'd like to stop NULL allocations as soon as one happens
- We can detect when a NULL allocation happens
  - Have CPU core 1 perform slab heap exhaustion
  - Have CPU core 0 monitor the NULL page for changes
    - We'll detect this assignment: `node->element = obj;`
- We can't stop new node allocations from happening…
- …but maybe we can stop them from being NULL!
  - Have CPU core 0 free some nodes as soon as it detects the NULL allocation
  - We can do this by signaling an object that another thread was waiting on

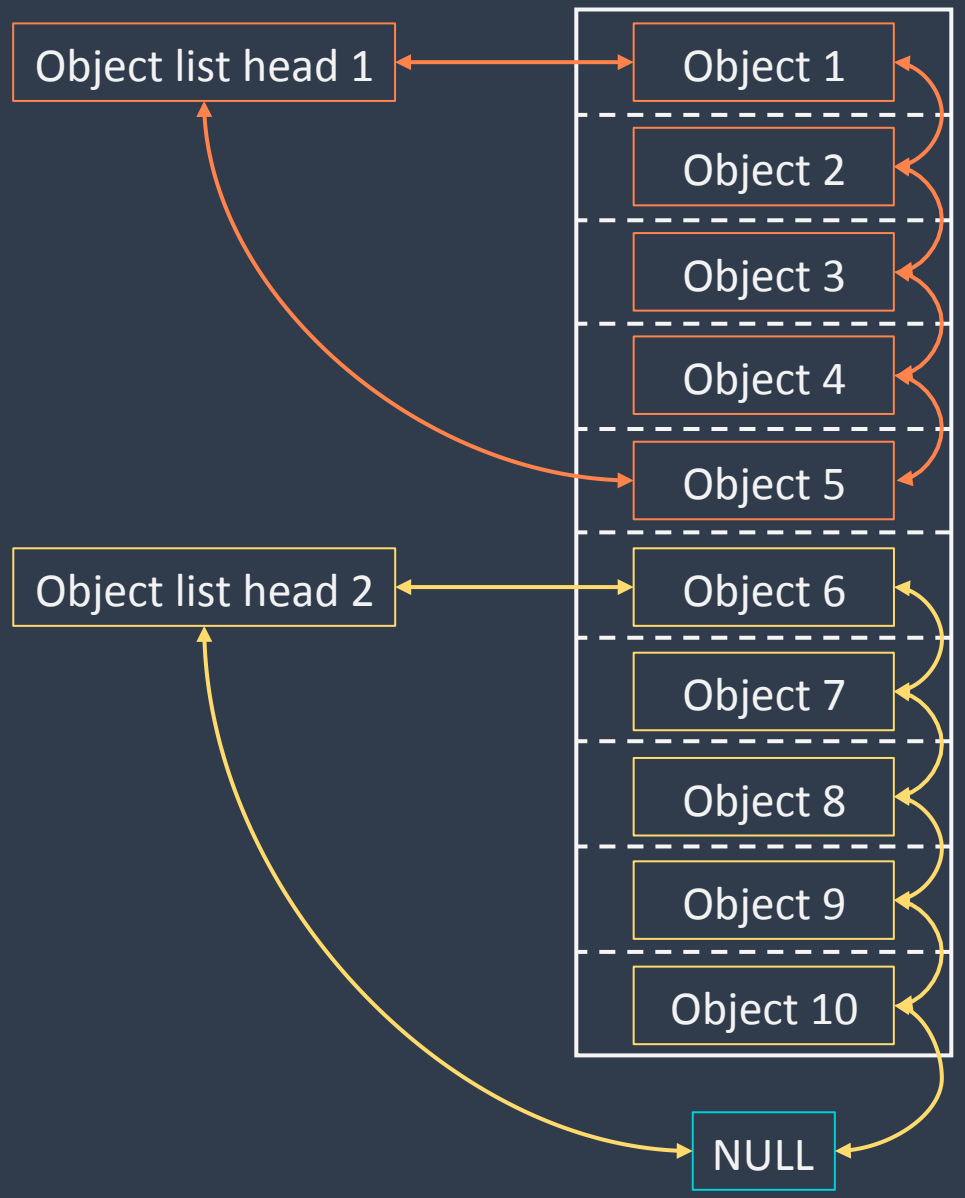
# Just-in-time node freeing

- Core 1 just exhausted the linked list node slab heap
- Core 0 sees a change on the NULL page

```
00000000 00000000 00000000 00000000
```

just became

```
nextptr  prevptr  objptr   00000000
```




Slab heap was just exhausted

# Just-in-time node freeing

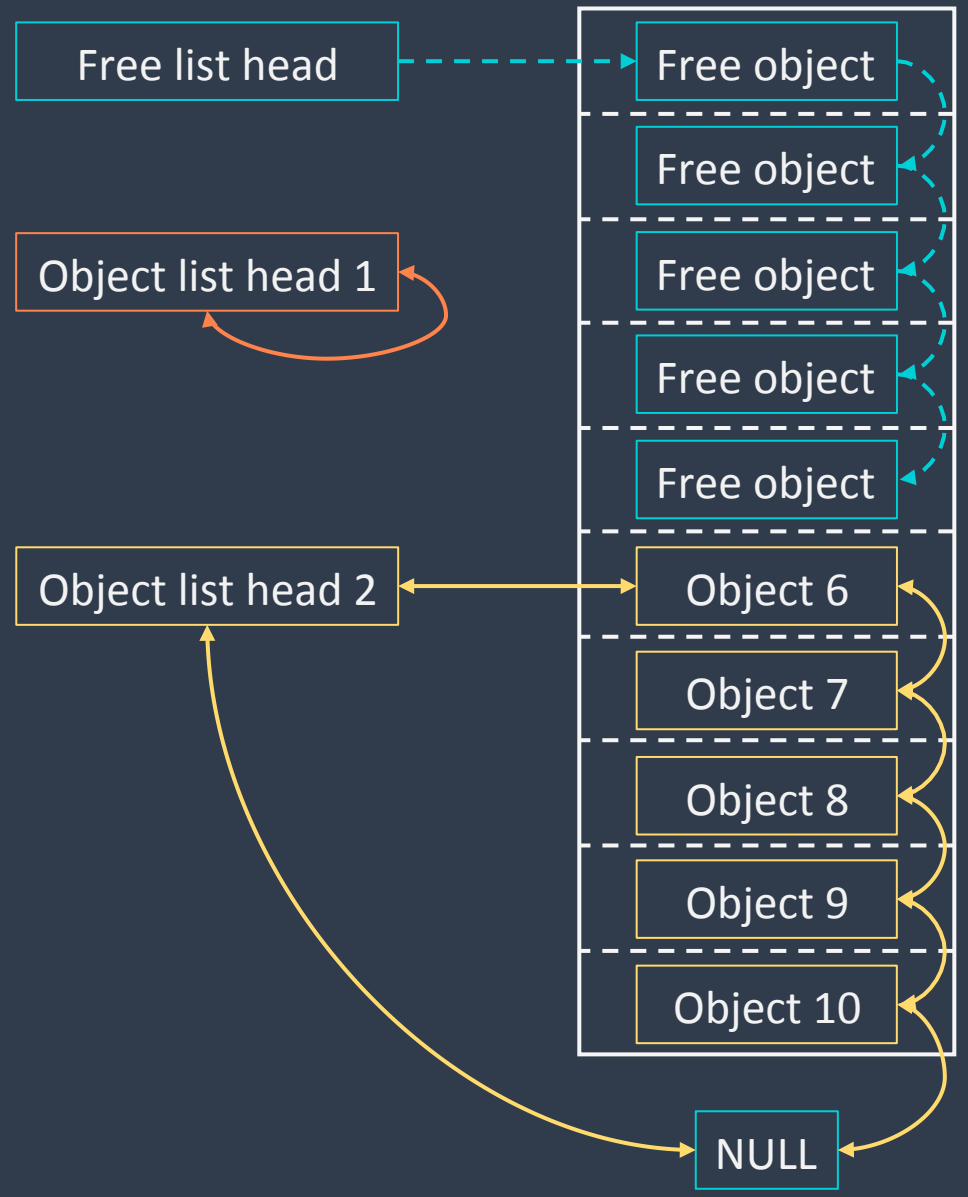
- Core 1 just exhausted the linked list node slab heap
- Core 0 sees a change on the NULL page

```
00000000 00000000 00000000 00000000
```

just became

```
nextptr  prevptr  objptr   00000000
```

- Core 0 calls svcSignalEvent to free a bunch of linked list nodes




Slab heap was just exhausted

# Just-in-time node freeing

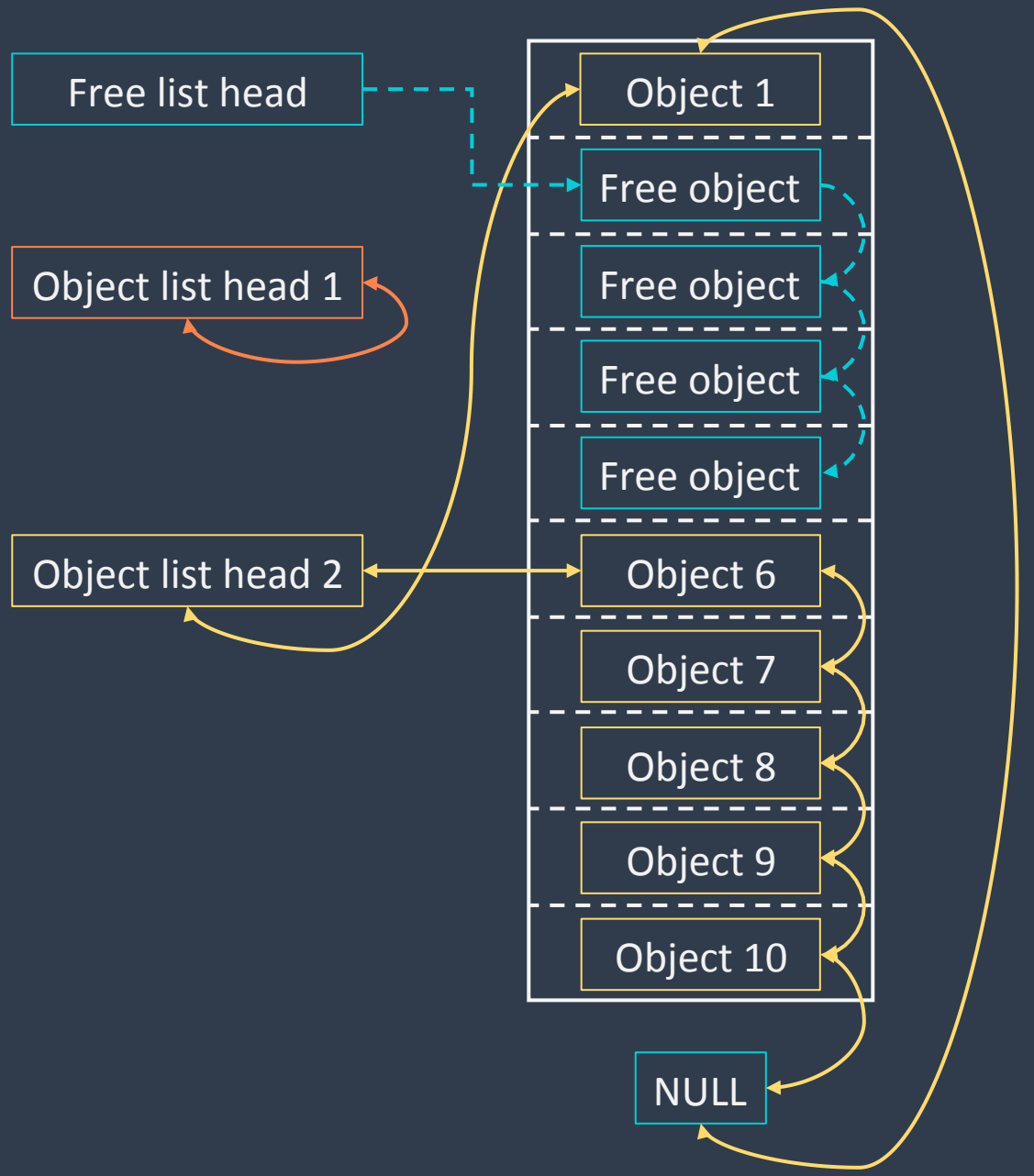
- Core 1 just exhausted the linked list node slab heap
- Core 0 sees a change on the NULL page

```
00000000 00000000 00000000 00000000
```

just became

```
nextptr  prevptr  objptr   00000000
```

- Core 0 calls svcSignalEvent to free a bunch of linked list nodes
- Next allocations use the free nodes as intended




Slab heap was just exhausted

# How do we get code execution?

- When the NULL node is unlinked, we control `node->next` and `node->prev`

=> We can write an arbitrary value to an arbitrary location

  - Has to be a **writable pointer** value…

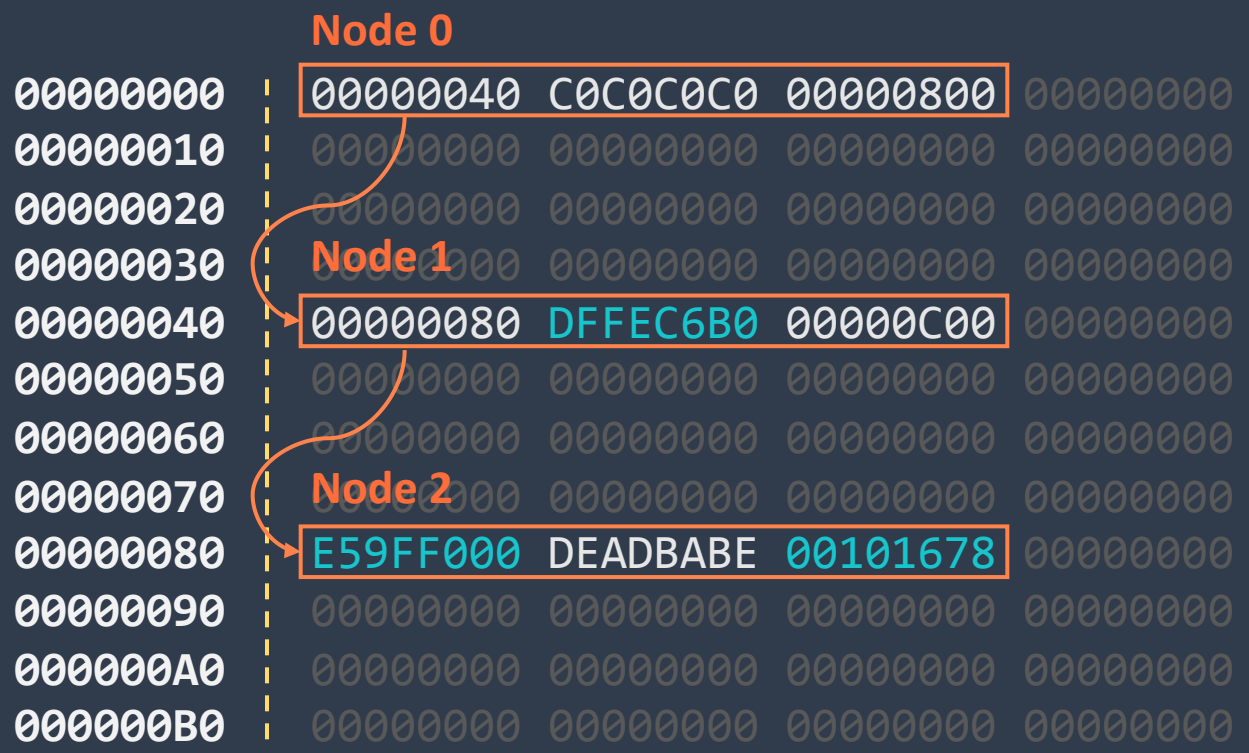
- But what to overwrite?
  - vtable is used immediately after unlinking for an indirect call…
- Difficulties
  - `free_kobj` kernel panics on NULL
  - Unlinking writes to our target and our value – so writing a code address is annoying

```
KLinkedList::remove:
...

KLinkedListNode *next = node->next;
KLinkedListNode *prev = node->prev;
next->prev = prev;
prev->next = next;
node->next = 0;
node->prev = 0;

...
--------------------------------

...

KLinkedList::remove(...);
free_kobj(&freelist_listnodes, node);
((int(*)(_DWORD, _DWORD))(vtable[9]))(...);

...
```

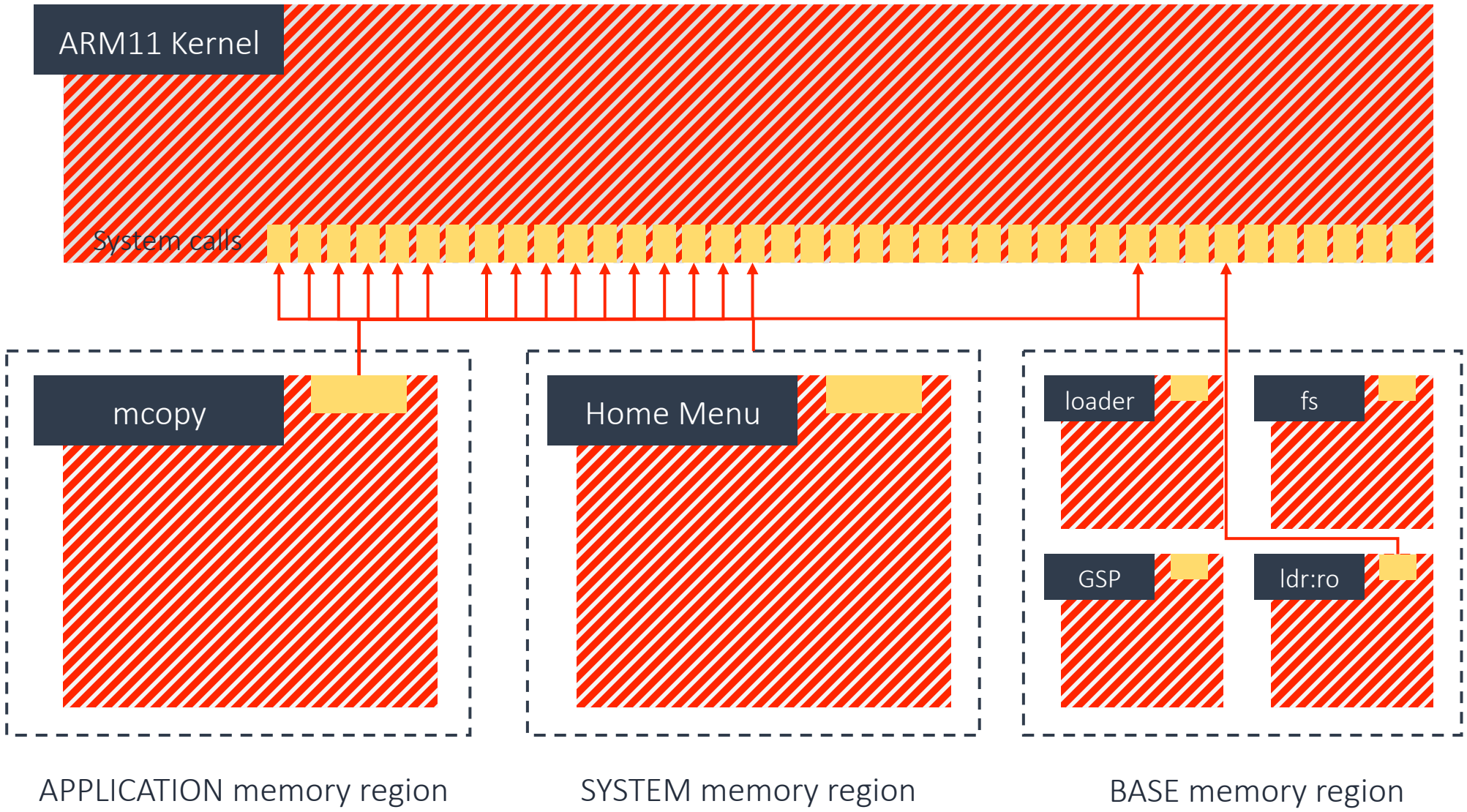
Linked list unlinking

# Manufacturing the linked list

- **Node 0**
  - **Next**: node 1
  - **Prev**: irrelevant (unused)
  - **Element**: fake object that *won't* trigger unlinking
- **Node 1**
  - **Next**: node 2
  - **Prev**: address of target vtable slot
  - **Element**: fake object that *will* trigger unlinking
- **Node 2**
  - **Next**: "`ldr pc, [pc]`"
  - **Prev**: irrelevant (unlink overwrites it)
  - **Element**: address loaded by "`ldr pc, [pc]`"




RWX NULL page linked list nodes

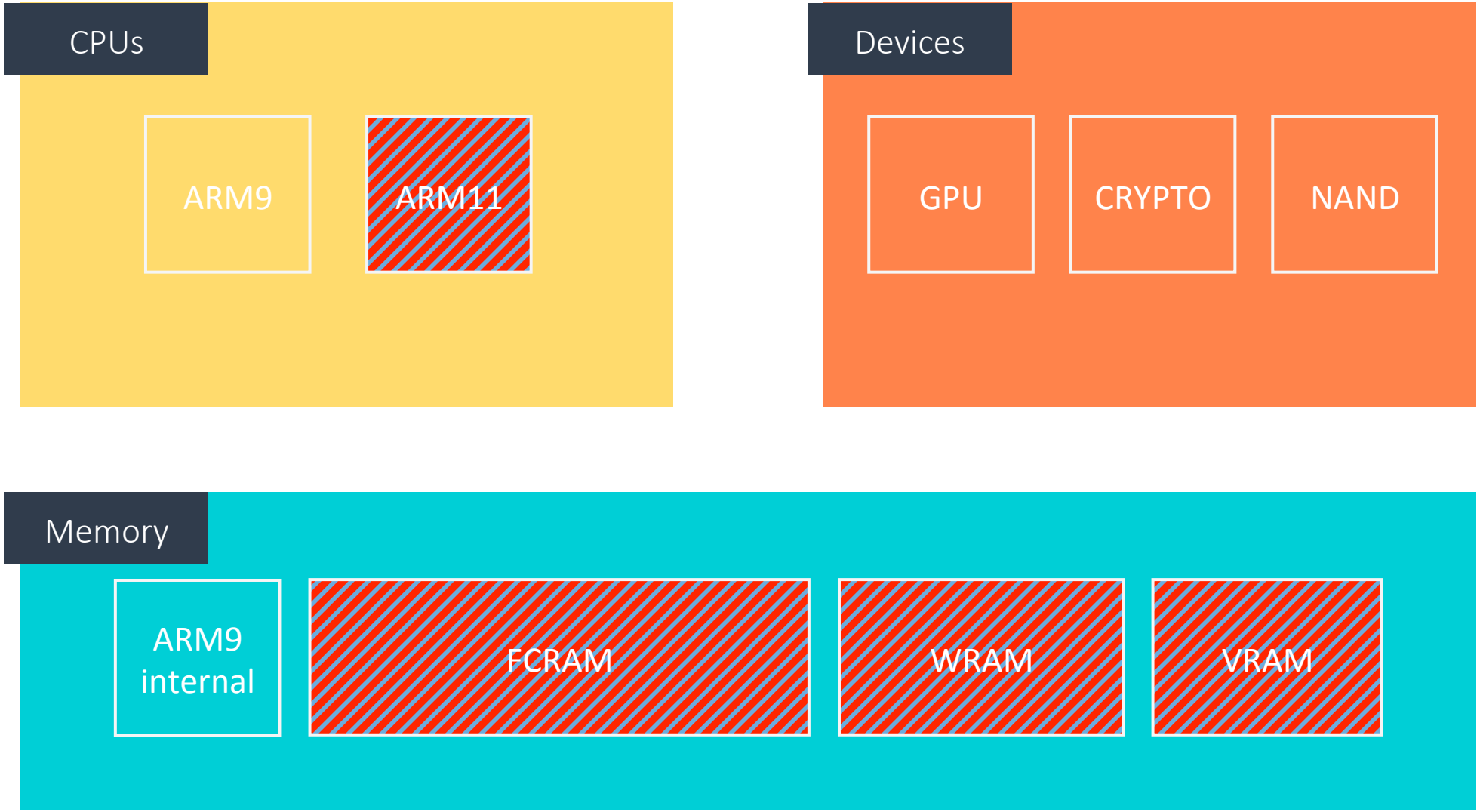



ARM11 kernel compromised, and therefore all ARM11 processes as well

# Taking over the ARM9

Because nothing’s ever enough with you people






# What's compromised so far

# ARM9 responsibilities

- Brokers access to storage (SD, NAND…)
  - Includes title management (installing, updating…)
- Decrypts and verifies/authenticates content
  - Owning ARM11 is enough to run pirated content, but not to decrypt new content if an update is released
- Handles backwards compatibility
  - How does that work?

## NATIVE_FIRM

- Main FIRM
- Runs apps and games
- 3DS boots it by default

## SAFE_FIRM

- “Safe mode” FIRM
- Used to do firmware updates

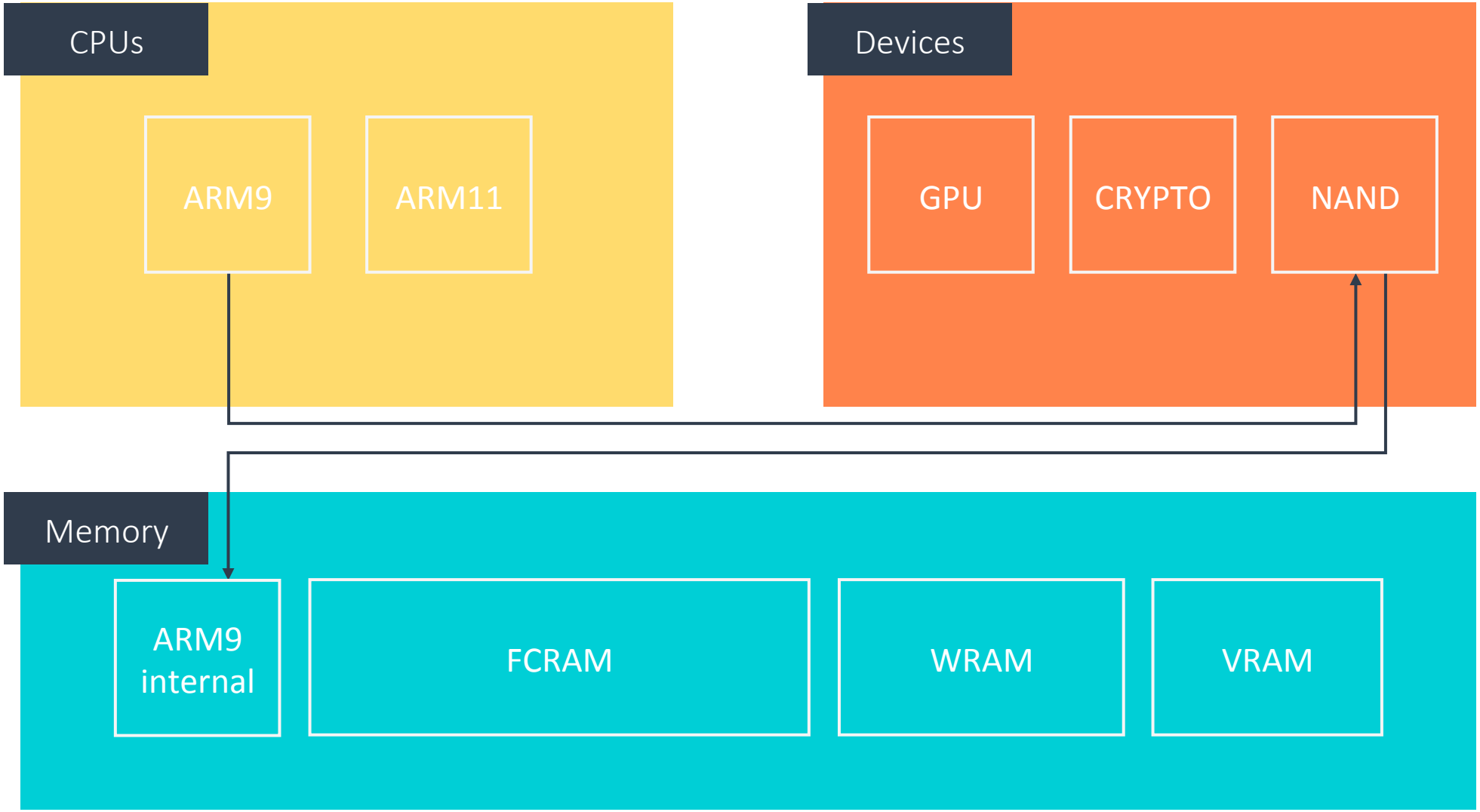
## TWL_FIRM

- TWL = DSi codename
- Used to run DS and DSi software

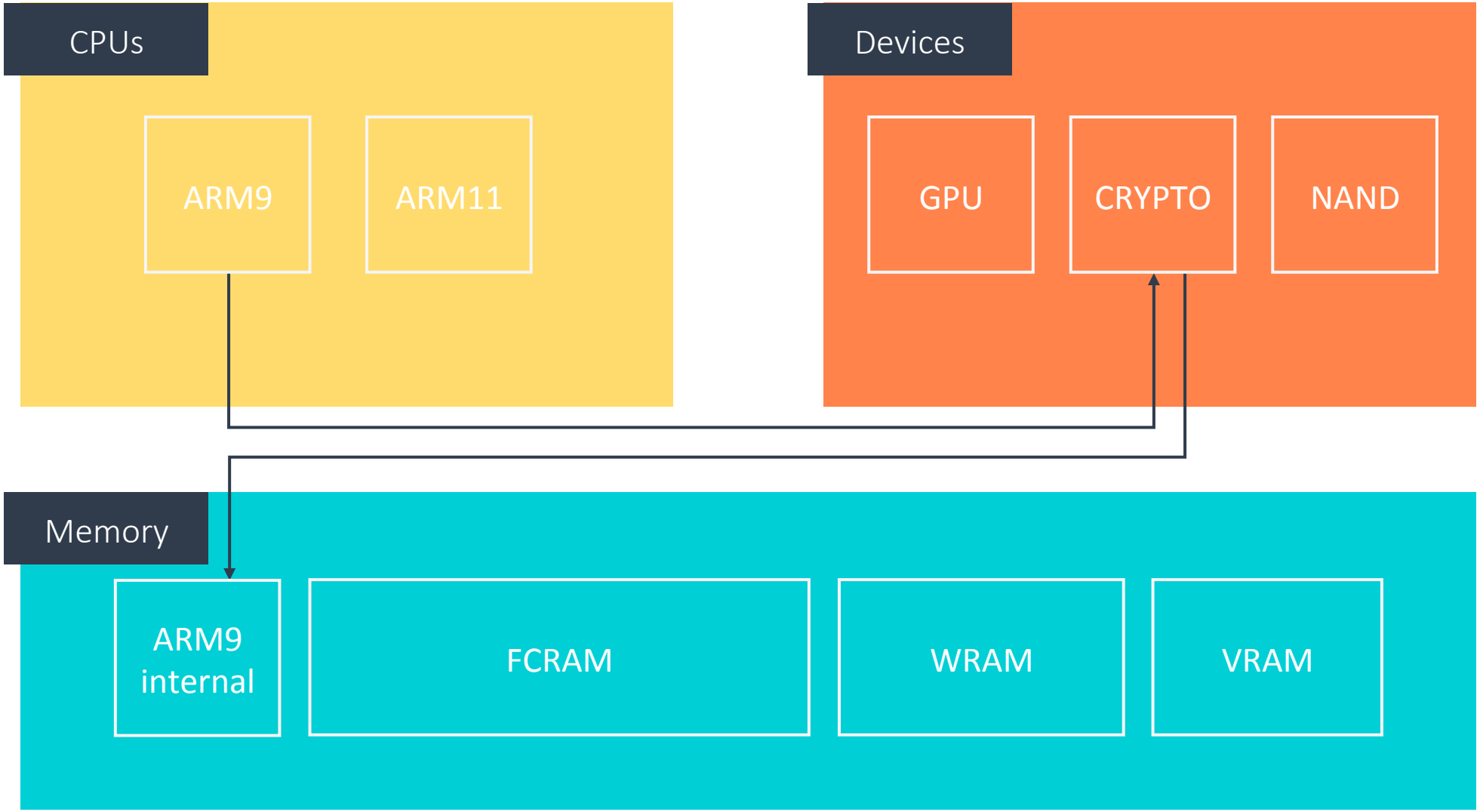
## AGB_FIRM

- AGB = GBA codename
- Used to run GBA software

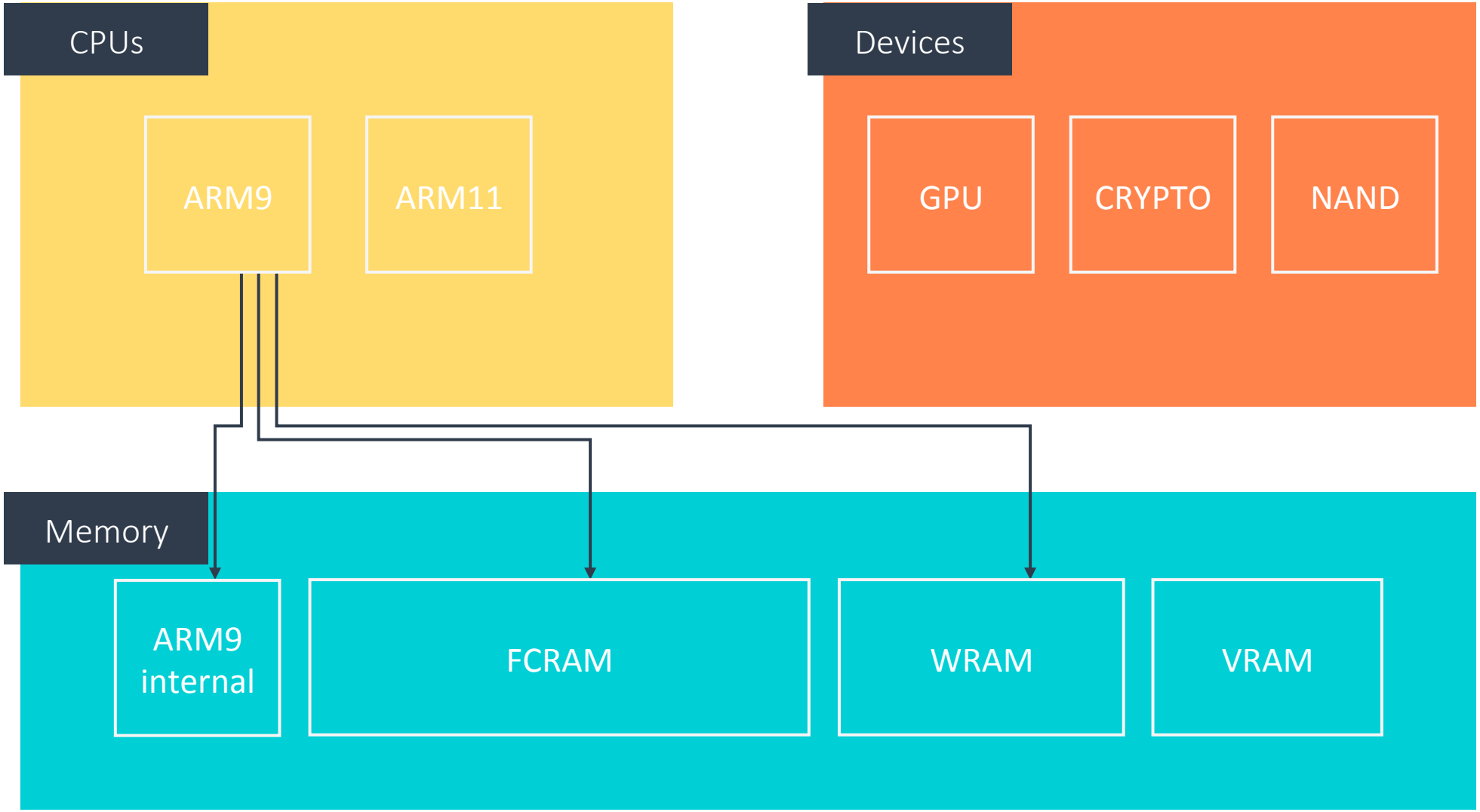
# The 3DS’s “FIRM” firmware system



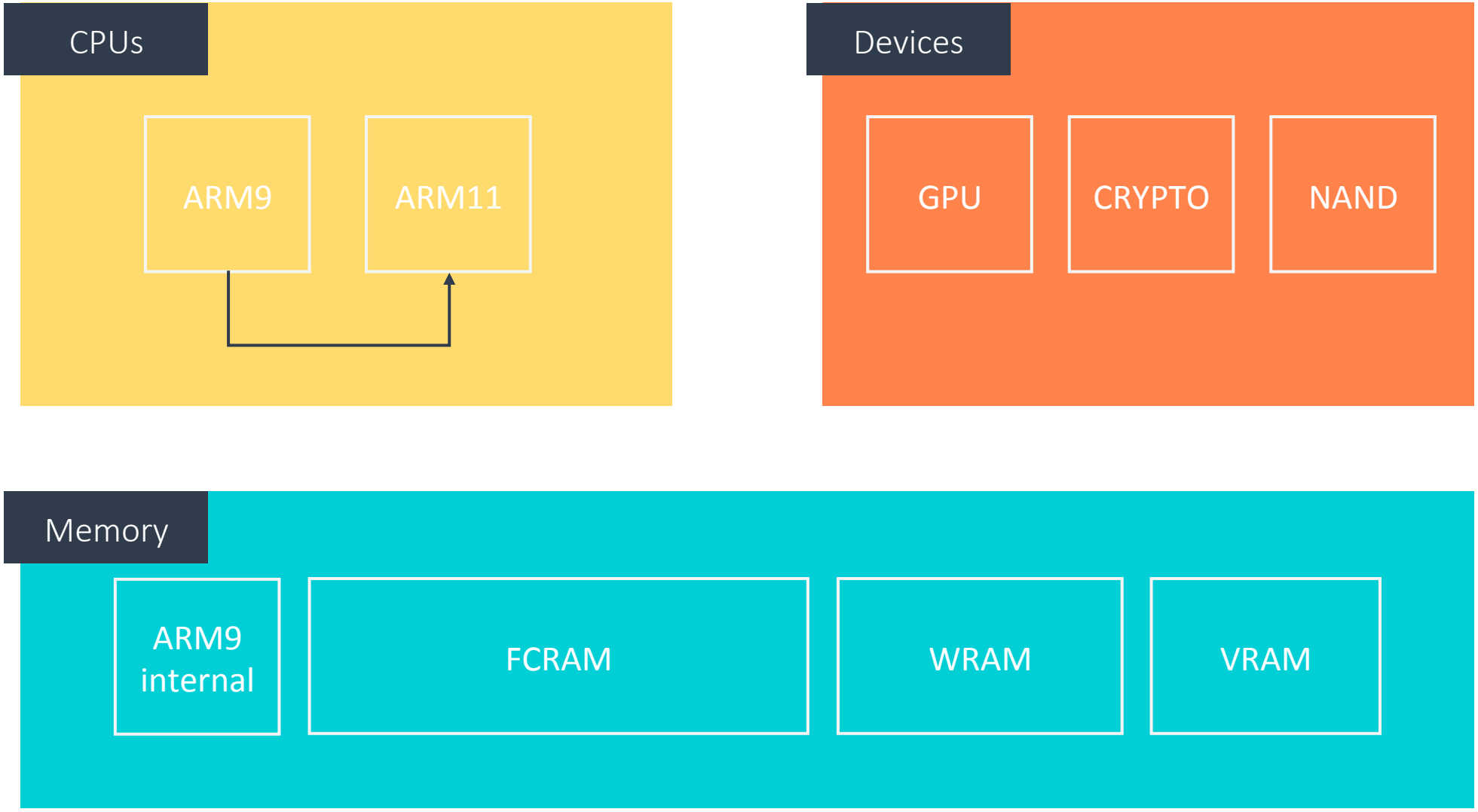

FIRM launch: ARM9 loads FIRM from NAND



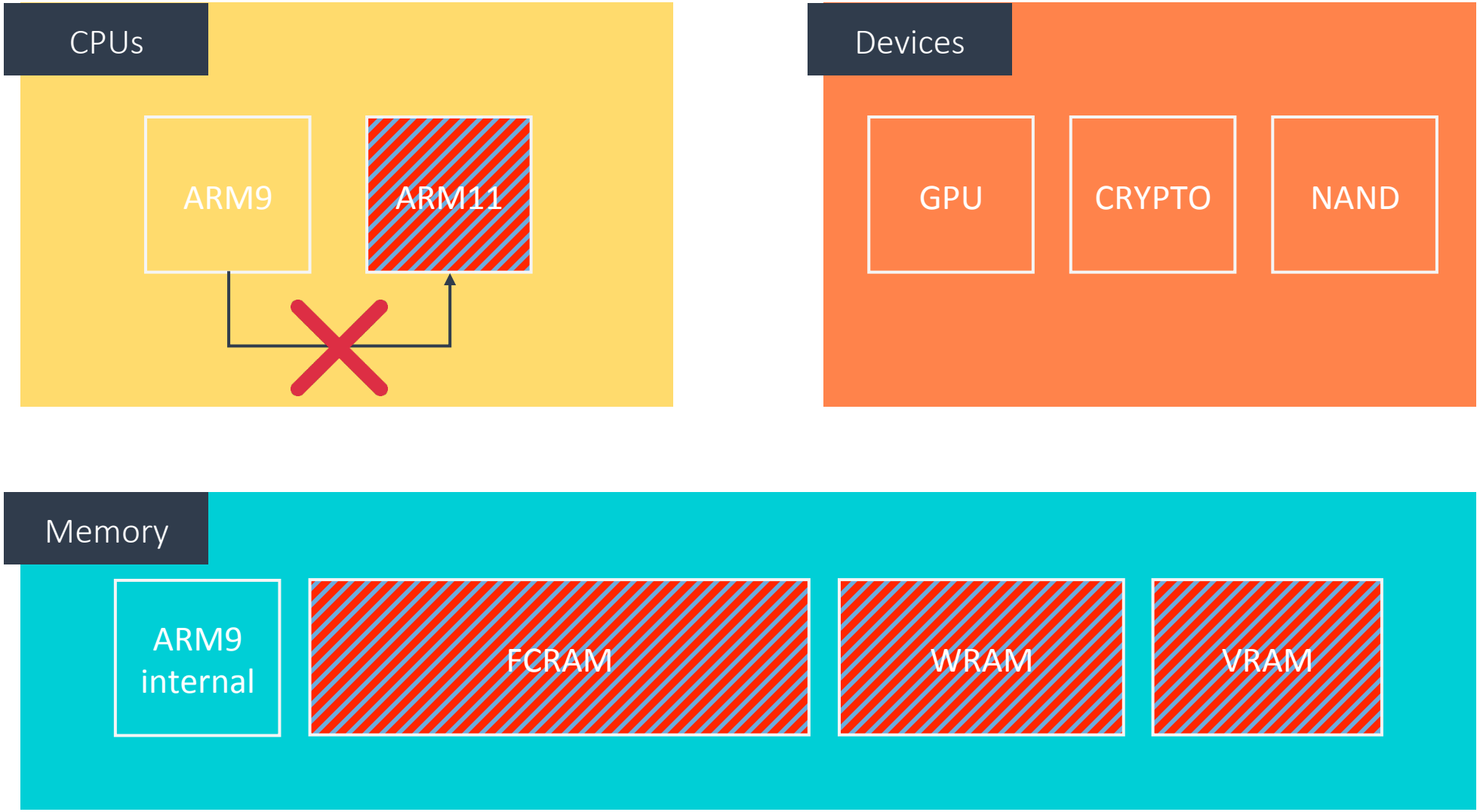

FIRM launch: ARM9 uses CRYPTO hardware to decrypt and authenticate FIRM



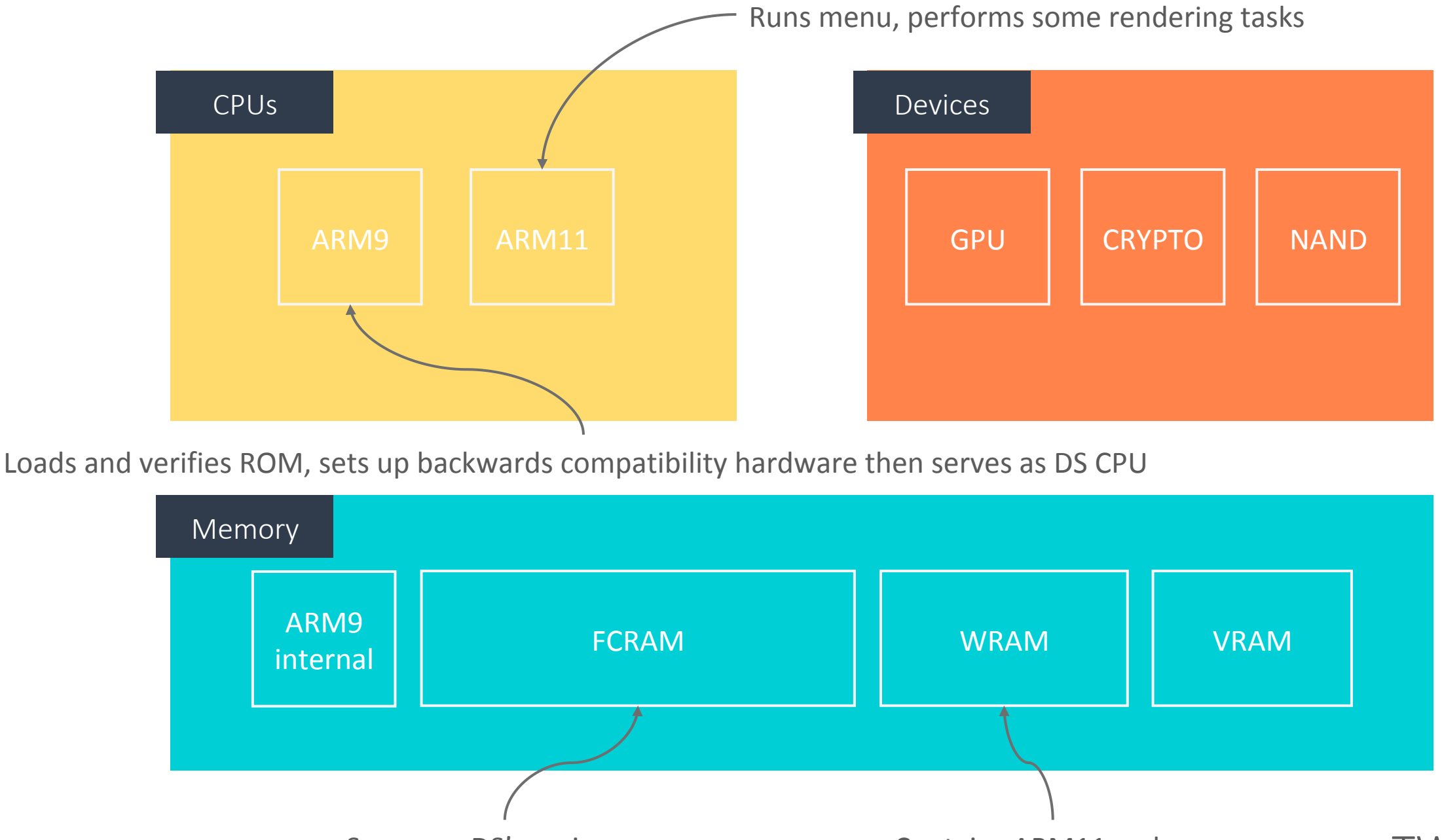

FIRM launch: ARM9 copies sections to relevant locations




FIRM launch: ARM9 signals ARM11 to run its FIRM section and then runs its own




FIRM launch: a compromised ARM11 can just keep running its own code


Runs menu, performs some rendering tasks
CPUs
ARM9
ARM11
Devices
GPU
CRYPTO
NAND
Loads and verifies ROM, sets up backwards compatibility hardware then serves as DS CPU
Memory
ARM9 internal
FCRAM
WRAM
VRAM
Serves as DS's main RAM
Contains ARM11 code

TWL_FIRM

# Where do ROMs come from?

- TWL_FIRM can load ROMs from multiple sources
  - Gamecarts (physical games)
  - NAND (DSiWare)
  - ARM11 (…?)
- ROMs are authenticated before being parsed
  - DSi games are RSA signed
  - DS games weren’t signed so a their content is hashed and a whitelist is used
- This should be fine…
  - But for some reason, those checks are bypassed when the ROM comes from the ARM11

8 bytes of 3DS address space == 2 bytes of DS space

```
3DS_PA = (NDS_PA - 0x02000000) * 4 + 0x20000000
```

If NDS_PA isn't properly bounded, then any 3DS_PA value is possible…

# DS mode memory layout

```
Header Overview
  Address Bytes Expl.
  000h    12    Game Title  (Uppercase ASCII, padded with 00h)
  00Ch    4     Gamecode    (Uppercase ASCII, NTR-<code>)         (0=homebrew)
  010h    2     Makercode   (Uppercase ASCII, eg. "01"=Nintendo) (0=homebrew)
  012h    1     Unitcode    (00h=Nintendo DS)
  013h    1     Encryption Seed Select (00..07h, usually 00h)
  014h    1     Devicecapacity         (Chipsize = 128KB SHL nn) (eg. 7 = 16MB)
  015h    9     Reserved               (zero filled)
  01Eh    1     ROM Version            (usually 00h)
  01Fh    1     Autostart (Bit2: Skip "Press Button" after Health and Safety)
                (Also skips bootmenu, even in Manual mode & even Start pressed)
  020h    4     ARM9 rom_offset    (4000h and up, align 1000h)
  024h    4     ARM9 entry_address (2000000h..23BFE00h)
  028h    4     ARM9 ram_address   (2000000h..23BFE00h)
  02Ch    4     ARM9 size          (max 3BFE00h) (3839.5KB)
  030h    4     ARM7 rom_offset    (8000h and up)
  034h    4     ARM7 entry_address (2000000h..23BFE00h, or 37F8000h..3807E00h)
  038h    4     ARM7 ram_address   (2000000h..23BFE00h, or 37F8000h..3807E00h)
  03Ch    4     ARM7 size          (max 3BFE00h, or FE00h) (3839.5KB, 63.5KB)
  ...
```

DS ROM header format (credit: gbatek https://problemkaputt.de/gbatek.htm)

No upper bound on `section_ram_address`

No integer overflow check

- `section_ram_address >= 0x02000000`
- `section_ram_address + section_size <= 0x02FFC000`
- `section doesn't intersect with [0x023FEE00; 0x023FF000]`
- `section doesn't intersect with [0x03FFF600; 0x03FFF800]`

No bounds check on `section_size`

# TWL_FIRM ROM loader code section checks

# What if we want to write to `0x0806E634`?

Example values:

- `section_ram_address = 0xBC01B98D`
- `section_size = 0x43FE4673`
- `(0xBC01B98D - 0x02000000) * 4 + 0x20000000 = 0x0806E634`

Constraints are respected:

- `0xBC01B98D >= 0x02000000`
- `0xBC01B98D + 0x43FE4673 = 0 <= 0x02FFC000`
- `section doesn't intersect with [0x023FEE00; 0x023FF000]`
  - Because `0xBC01B98D > 0x023FF000`
- `section doesn't intersect with [0x03FFF600; 0x03FFF800]`
  - Because `0xBC01B98D > 0x03FFF800`

# What about the huge section size ?

- We have section_size = 0x43FE4673
- 0x43FE4673 bytes is about 1GB of data
- => we will crash if we can't interrupt the copy while it's happening…
- Fortunately, load_nds_section copies in blocks of 0x10000 bytes at most

Performs the actual copy – can hijack its return address

```
void load_nds_section(u32 ram_address, u32 rom_offset, u32 size, ...)
{
  ...

  u32 rom_endoffset = rom_offset + size;
  u32 rom_offset_cursor = rom_offset;
  u32 ndsram_cursor = ram_address;

  while ( rom_offset_cursor < rom_endoffset )
  {
    curblock_size = 0x10000;
    if ( rom_endoffset - rom_offset_cursor < curblock_size )
    {
      curblock_size = align32(rom_endoffset - rom_offset_cursor);
    }

    memcpy(buf, rom_offset_cursor + 0x27C00000, curblock_size);

    ...

    write_ndsram_section(ndsram_cursor, buf, curblock_size);

    rom_offset_cursor += curblock_size;
    ndsram_cursor += curblock_size;
  }

  ...
}
```

ROM section loading code

```
void write_ndsram_section(u32 ndsram_dst, u16* src, int len)
{
  u16* ctr_pa_dst = convert_ndsram_to_ctrpa(ndsram_dst);

  for(int i = len; i != 0; i -= 2)
  {
    *ctr_pa_dst = *src;

    ctr_pa_dst += 4;
    src += 4;
  }
}
```

Copies 2 bytes at a time…

…every 8 bytes

# TWL_FIRM's "weird" memcpy

load_nds_section stack

| Address | Value |
|---|---|
| 0806E634 | 08032F41 |
| 0806E638 | 00000000 |
| 0806E63C | 080C0000 |
| 0806E640 | C0180000 |
| 0806E644 | 08033851 |
| 0806E648 | 00000000 |
| 0806E64C | 00010000 |
| 0806E650 | 0806E66C |
| 0806E654 | 00000001 |
| 0806E658 | 00010000 |
| 0806E65C | 0808922C |
| 0806E660 | 00010000 |
| 0806E664 | 08089E64 |
| 0806E668 | 0808923C |
| 0806E66C | 0803DCDC |

`write_ndsram_section` return address

Bytes we can overwrite

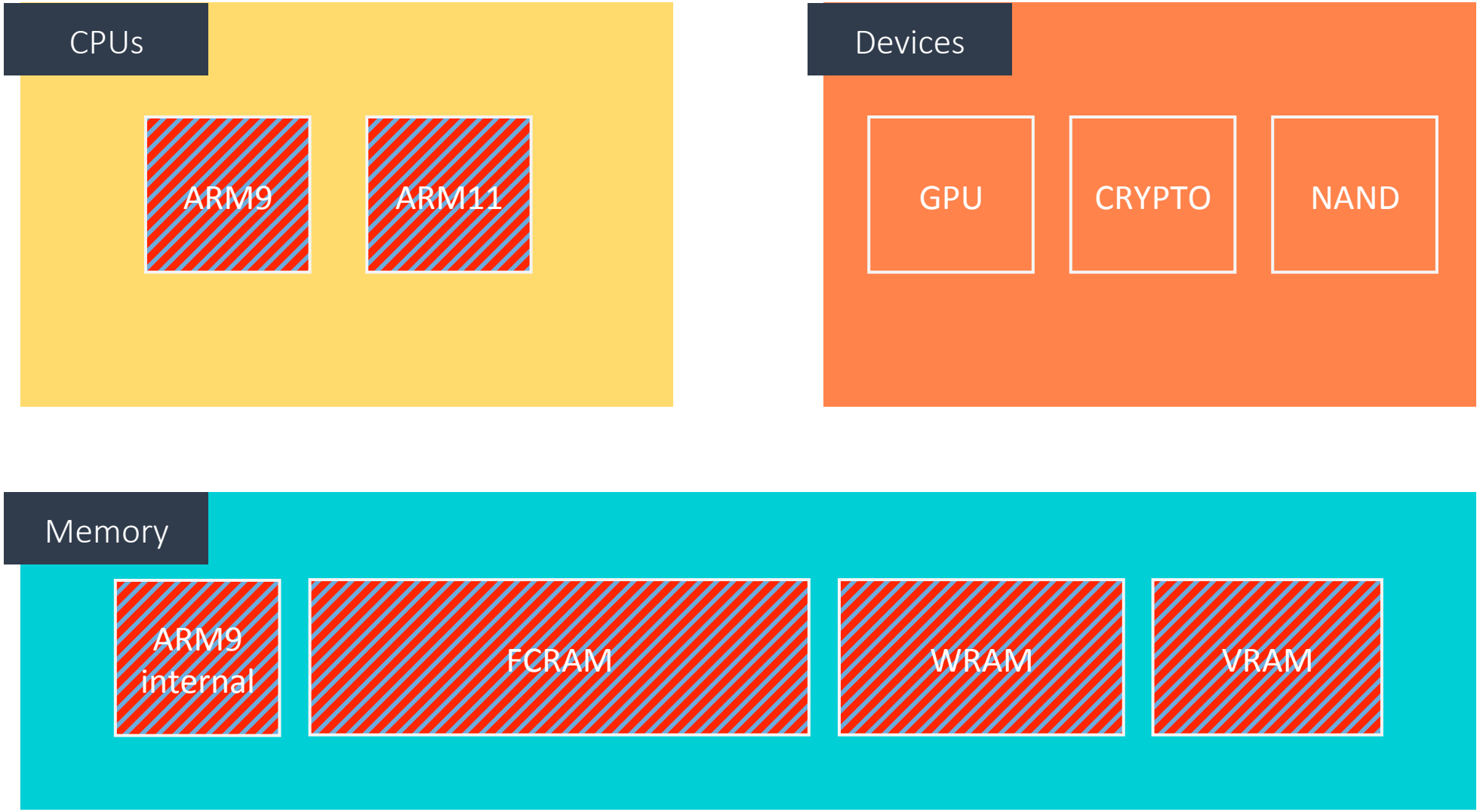
⇒ We can only redirect to a gadget within a 0x10000 byte region

⇒ We can only generate addresses within 0x10000 byte regions determined by pointers already on the stack

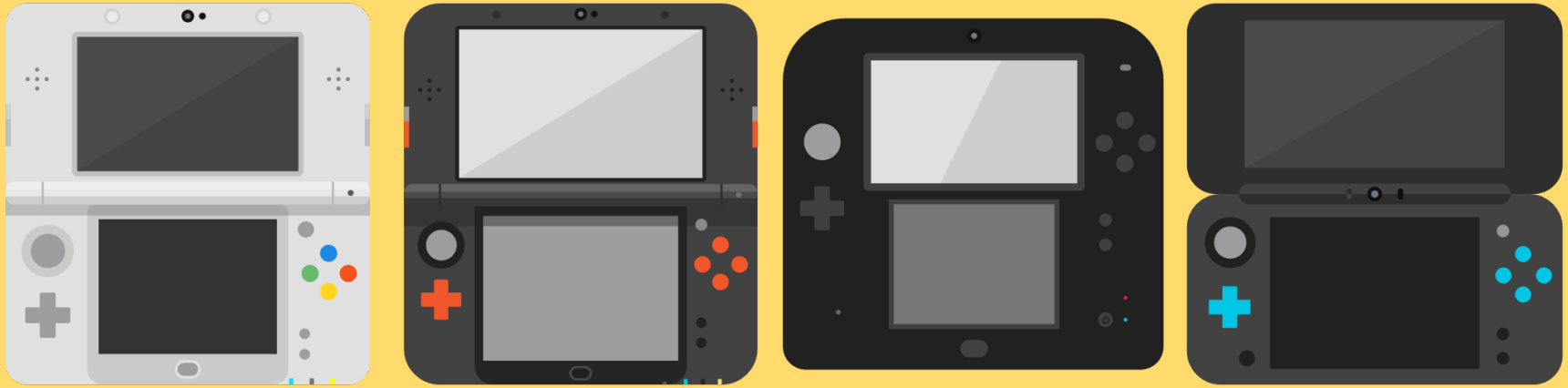
Corrupting the stack

# load_nds_section stack

| Address | Value | |
|---|---|---|
| 0806E634 | 08035512 | `ADD SP, SP, #0x14` / `POP {R4-R7,PC}` |
| 0806E638 | 00000000 | |
| 0806E63C | 080C0000 | |
| 0806E640 | C0180000 | `ADD SP, SP, #0x14` |
| 0806E644 | 08033851 | |
| 0806E648 | 00000000 | |
| 0806E64C | 00010000 | |
| 0806E650 | 0806E66C | |
| 0806E654 | 00000001 | `POP {R4-R7}` |
| 0806E658 | 00010000 | |
| 0806E65C | 08089064 | Points to code in the NDS ROM header (Process9 doesn't have DEP) |
| 0806E660 | 00010000 | |
| 0806E664 | 08089E64 | |
| 0806E668 | 0808923c | |
| 0806E66C | 0803DCDC | |


CPUs
ARM9
ARM11
Devices
GPU
CRYPTO
NAND
Memory
ARM9 internal
FCRAM
WRAM
VRAM

ARM9 down ☺



Code available at github.com/smealum

## Thanks to:

derrek, nedwill, yellows8, plutoo, naehrwert

@smealum

# Icon credits

- https://www.webdesignerdepot.com/2017/07/free-download-flat-nintendo-icons/
- https://www.flaticon.com/free-icon/gaming_771247
- https://www.flaticon.com/free-icon/checked_291201
- https://www.flaticon.com/free-icon/close_579006
- https://www.flaticon.com/free-icon/twitter_174876