EMULATING HYPERVISORS: A SAMSUNG RKP CASE STUDY

ARISTEIDIS THALLAS (@_athallas)

athallas@census-labs.com

OFFENSIVE CON 2020

www.census-labs.com

▷ $ whoami

- Electrical & Computer Engineer
- Used to build robots academically
- Security researcher at CENSUS S.A.
- Vulnerability Research, Reverse Engineering, Exploit Development, etc.
- Used to break virtualization software
- Now mostly breaking Androids

▷ AGENDA

- Problem statement
- Introduction
  - ARM architecture & virtualization extensions
  - Samsung hypervisor
- Framework implementation & RKP analysis
  - System boot
    - EL1 initialization
  - Hypervisor boot
    - Hypervisor boot termination
  - Hypervisor initialization
    - Hypervisor exception handling
    - Hypervisor initialization
  - Demo
- Fuzzing
  - Dummy fuzzing
- Conclusions
- References

▷ Problem Statement

- Kernel != last line of defense against full system compromise
  - Various protections
  - ARM Virtualization Extensions (VE) utilized for runtime kernel protection in Android ecosystem
- Android debugging is an uphill struggle
  - Critical in RE, vuln research and exploit dev

▷ Problem Statement

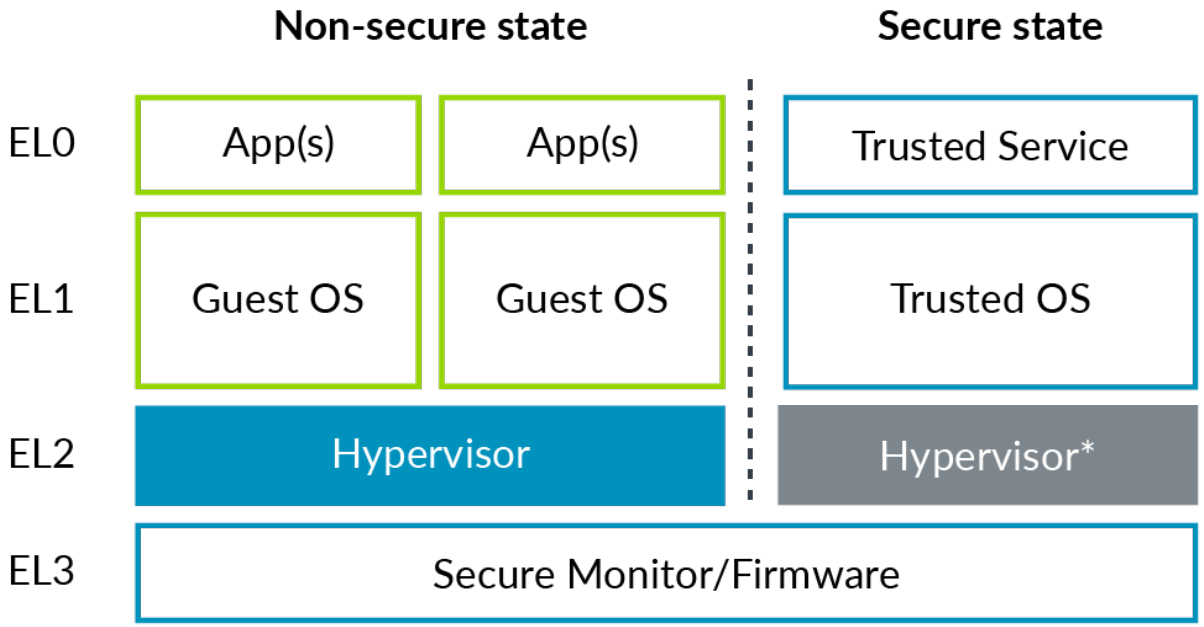
- Emulate components under QEMU
  - System observability
- Discuss
  - ARM low level concepts
  - Virtualization extensions
- Demonstrate via minimal framework for Samsung RKP
- Investigate fuzzing setups
- Build you own frameworks ;)

▷ AGENDA

- Problem statement
- Introduction
  - **ARM architecture & virtualization extensions**
  - Samsung hypervisor
- Framework implementation & RKP analysis
  - System boot
    - EL1 initialization
  - Hypervisor boot
    - Hypervisor boot termination
  - Hypervisor initialization
    - Hypervisor exception handling
    - Hypervisor initialization
  - Demo
- Fuzzing
  - Dummy fuzzing
- Conclusions
- References

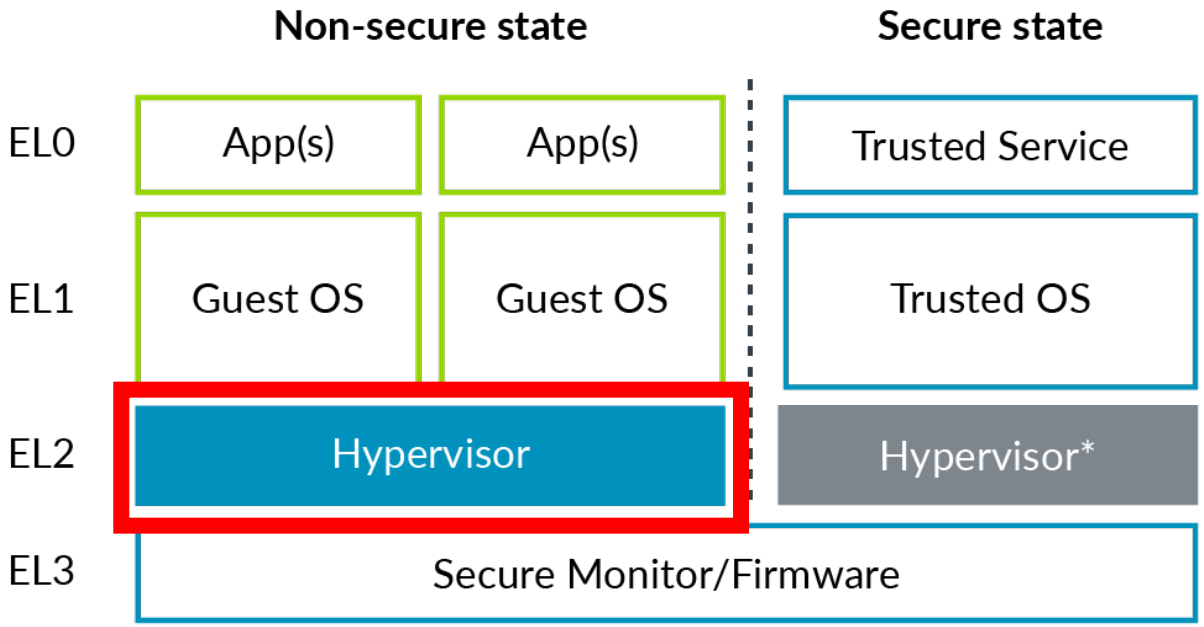
▷ ARM Arch & Virtualization Exts

- 2 Security states (secure, non-secure)
- 4 exception levels (ELs) aka Execution Levels
  - svc, hvc, smc, eret



▷ ARM Arch & Virtualization Exts

- 2 Security states (secure, non-secure)
- 4 exception levels (ELs) aka Execution Levels
  - svc, hvc, smc, eret

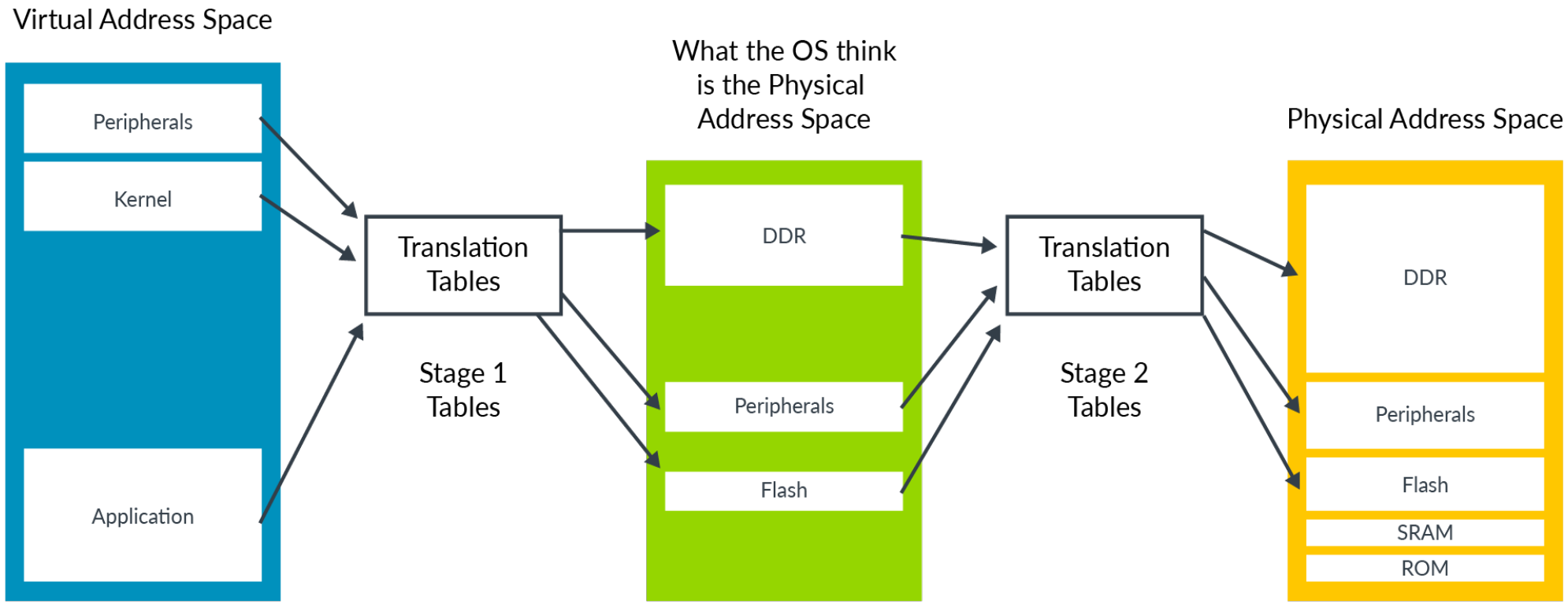


▷ ARM Arch & Virtualization Exts

- Hypervisor – EL2
  - Dictate EL1 behavior
- Hypervisor Configuration Register (HCR_EL2)
  - Access to EL1 registers, etc.
- Stage 2 translation
  - Disabled: EL1 VA -> PA
  - Enabled: EL1 VA -> IPA -> PA

▷ ARM Arch & Virtualization Exts

- Stage 2 translation graph



▷ AGENDA

- Problem statement
- Introduction
  - ARM architecture & virtualization extensions
  - **Samsung hypervisor**
- Framework implementation & RKP analysis
  - System boot
    - EL1 initialization
  - Hypervisor boot
    - Hypervisor boot termination
  - Hypervisor initialization
    - Hypervisor exception handling
    - Hypervisor initialization
  - Demo
- Fuzzing
  - Dummy fuzzing
- Conclusions
- References

▷ Samsung Hypervisor

- Security platform – Samsung KNOX
- Hypervisor – Realtime Kernel Protection (RKP)
- Targeted before
  - Samsung Galaxy S7 – Project Zero
- Focus on S8/Note8 implementation
  - Stripped and string obfuscation
  - Simple – allow focus on desired features
  - Arbitrary selection ¯_(ツ)_/¯

▷ AGENDA

- Problem statement
- Introduction
  - ARM architecture & virtualization extensions
  - Samsung hypervisor
- **Framework implementation & RKP analysis**
  - System boot
    - EL1 initialization
  - Hypervisor boot
    - Hypervisor boot termination
  - Hypervisor initialization
    - Hypervisor exception handling
    - Hypervisor initialization
  - Demo
- Fuzzing
  - Dummy fuzzing
- Conclusions
- References

▷ Framework Implementation

- Minimal implementation – EL3 & EL1
- ELF Aarch64 – kernel.elf
  - Supported by QEMU – simplified boot process
- Starting PA 0x80000000 (discussed later)
- QEMU configuration
  - virt platform
  - CPU cortex-a57
  - Single core/thread
  - 3GB RAM (discussed later)
  - Enabled Virtualization (EL2) & Secure (EL3) modes
  - Wait and attach to gdb

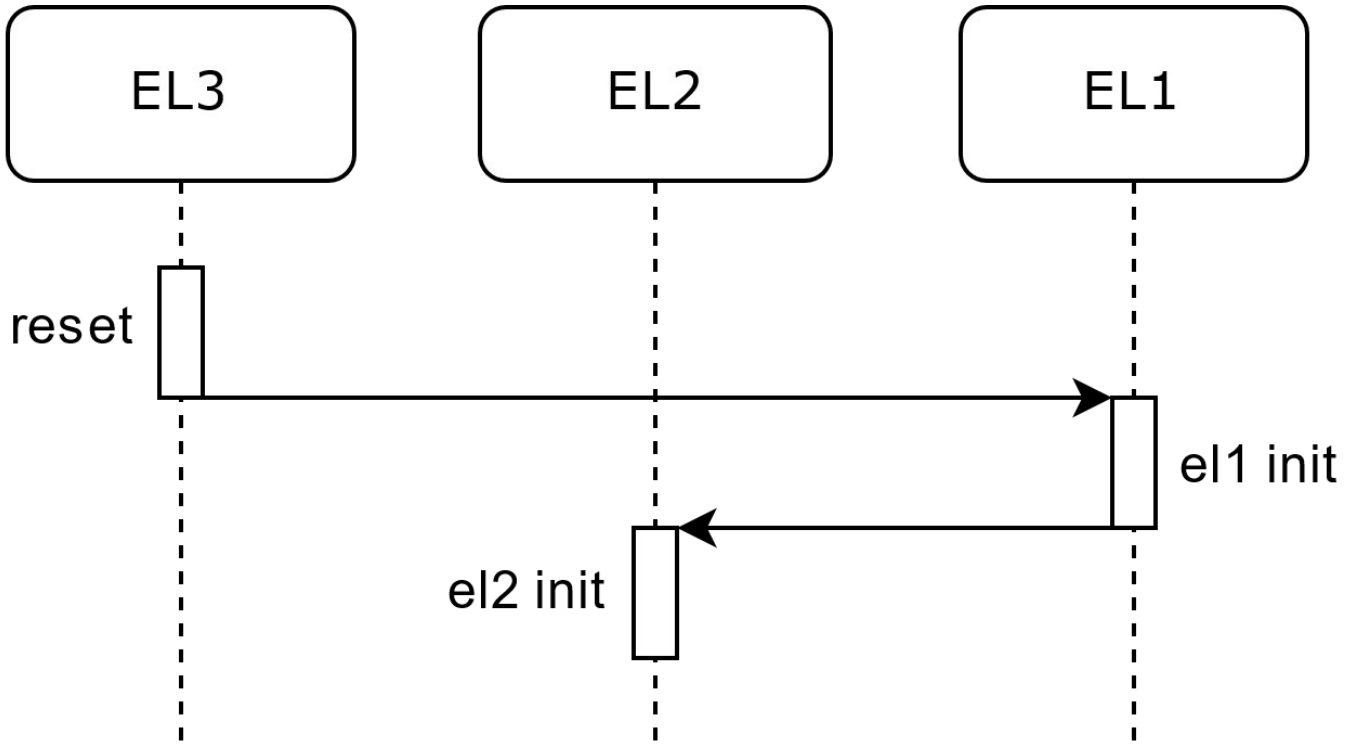
▷ Framework Implementation

```
$ qemu-system-aarch64                \
    -machine virt                    \
    -cpu cortex-a57                  \
    -smp 1                           \
    -m 3G                            \
    -kernel kernel.elf               \
    -machine gic-version=3           \
    -machine secure=true             \
    -machine virtualization=true     \
    -nographic                       \
    -S -s
```

```
$ aarch64-eabi-linux-gdb kernel.elf -q
    Reading symbols from kernel.elf...done.
    (gdb) target remote :1234
    Remote debugging using :1234
    _Reset () at boot64.S:15
    15          ldr x30, =stack_top_el3
    (gdb) disassemble
    Dump of assembler code for function _Reset:
    => 0x0000000080000000 <+0>:     ldr     x30, 0x80040000
       0x0000000080000004 <+4>:     mov     sp, x30
    ...
```

▷ Framework Implementation

- Framework high level approach



▷ AGENDA

- Problem statement
- Introduction
  - ARM architecture & virtualization extensions
  - Samsung hypervisor
- Framework implementation & RKP analysis
  - **System boot**
    - EL1 initialization
  - Hypervisor boot
    - Hypervisor boot termination
  - Hypervisor initialization
    - Hypervisor exception handling
    - Hypervisor initialization
  - Demo
- Fuzzing
  - Dummy fuzzing
- Conclusions
- References

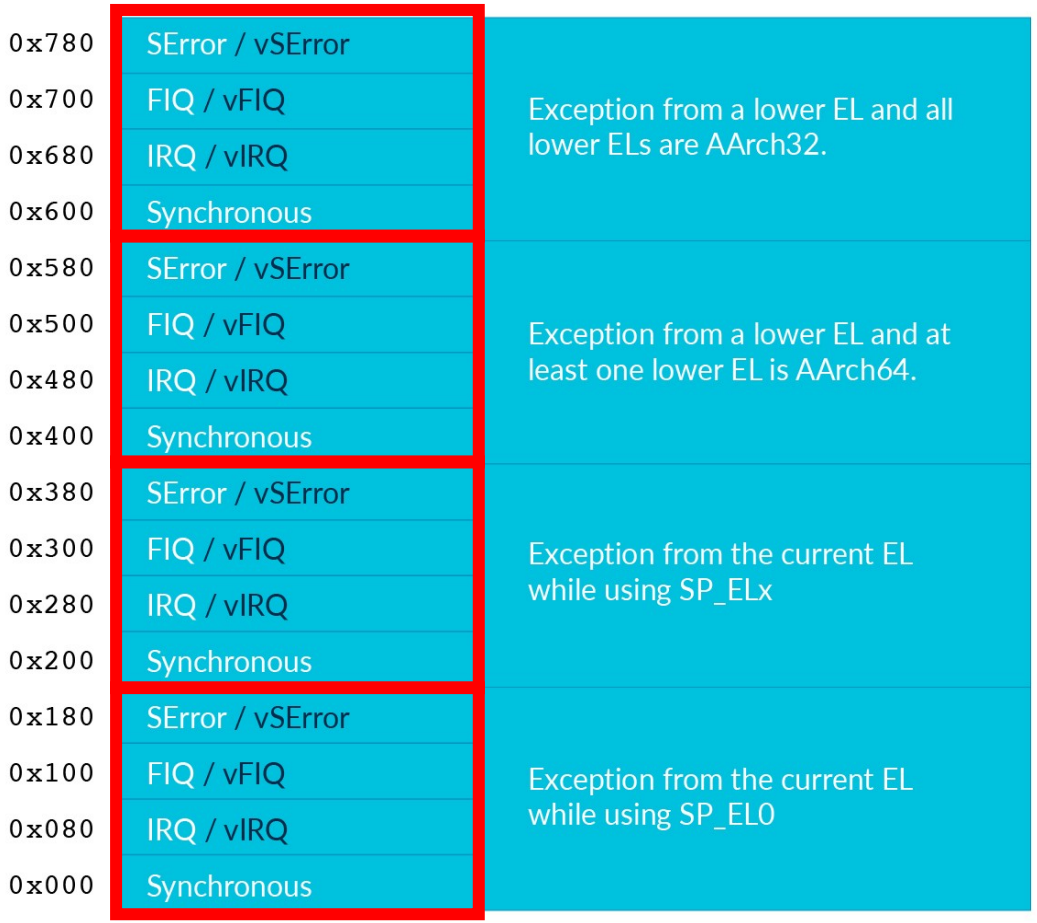
▷ System Boot

- Stack pointers – preallocated locations
- Exception vectors – VBAR_ELn
  - IRQs & FIQs disabled
  - No EL0
  - EL1 empty (inf loops)
  - EL3 only synchronous from lower AArch64

0x780	SError / vSError	Exception from a lower EL and all lower ELs are AArch32.
0x700	FIQ / vFIQ	
0x680	IRQ / vIRQ	
0x600	Synchronous	
0x580	SError / vSError	Exception from a lower EL and at least one lower EL is AArch64.
0x500	FIQ / vFIQ	
0x480	IRQ / vIRQ	
0x400	Synchronous	
0x380	SError / vSError	Exception from the current EL while using SP_ELx
0x300	FIQ / vFIQ	
0x280	IRQ / vIRQ	
0x200	Synchronous	
0x180	SError / vSError	Exception from the current EL while using SP_EL0
0x100	FIQ / vFIQ	
0x080	IRQ / vIRQ	
0x000	Synchronous	

▷ System Boot

- Stack pointers – preallocated locations
- Exception vectors – VBAR_ELn
  - IRQs & FIQs disabled
  - No EL0
  - EL1 empty (inf loops)
  - EL3 only synchronous from lower AArch64



▷ System Boot

- Stack pointers – preallocated locations
- Exception vectors – VBAR_ELn
  - IRQs & FIQs disabled
  - No EL0
  - EL1 empty (inf loops)
  - EL3 only synchronous from lower AArch64

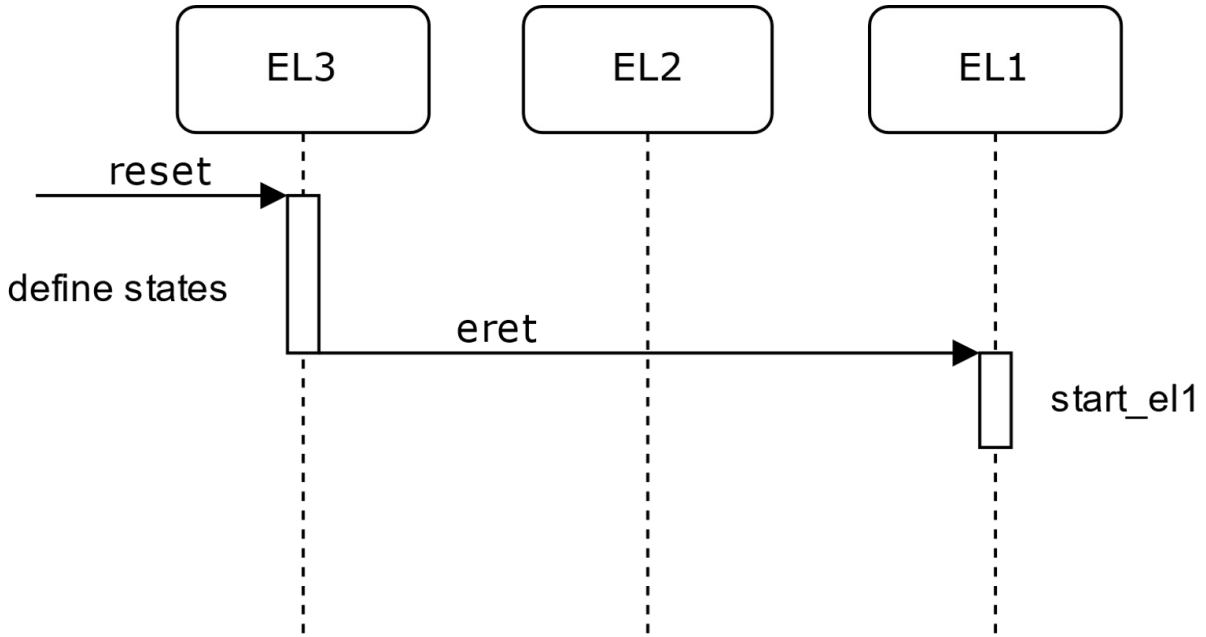
0x780	SError / vSError	Exception from a lower EL and all lower ELs are AArch32.
0x700	FIQ / vFIQ	
0x680	IRQ / vIRQ	
0x600	Synchronous	
0x580	SError / vSError	Exception from a lower EL and at least one lower EL is AArch64.
0x500	FIQ / vFIQ	
0x480	IRQ / vIRQ	
0x400	Synchronous	
0x380	SError / vSError	Exception from the current EL while using SP_ELx
0x300	FIQ / vFIQ	
0x280	IRQ / vIRQ	
0x200	Synchronous	
0x180	SError / vSError	Exception from the current EL while using SP_EL0
0x100	FIQ / vFIQ	
0x080	IRQ / vIRQ	
0x000	Synchronous	

▷ System Boot

- System registers – only EL3 & EL1 (almost ;)
- Start at EL3
- Secure Configuration Register (SCR_EL3)
  - SCR_EL3.NS – normal world
- Define EL2 state
  - SCR_EL3.RW – 64bit EL2
- Define EL1 state – HCR_EL2
  - HCR_EL2.RW – 64bit EL1
- If not set properly, cannot drop to lower EL

▷ System Boot

- Drop to EL1 by returning from a fake exception
  - Exception Link (ELR_EL3) – desired function
  - Saved Process Status (SPSR_EL3) – from EL1, AArch64
  - Exception Syndrom (ESR_EL3) – irrelevant
  - eret



▷ AGENDA

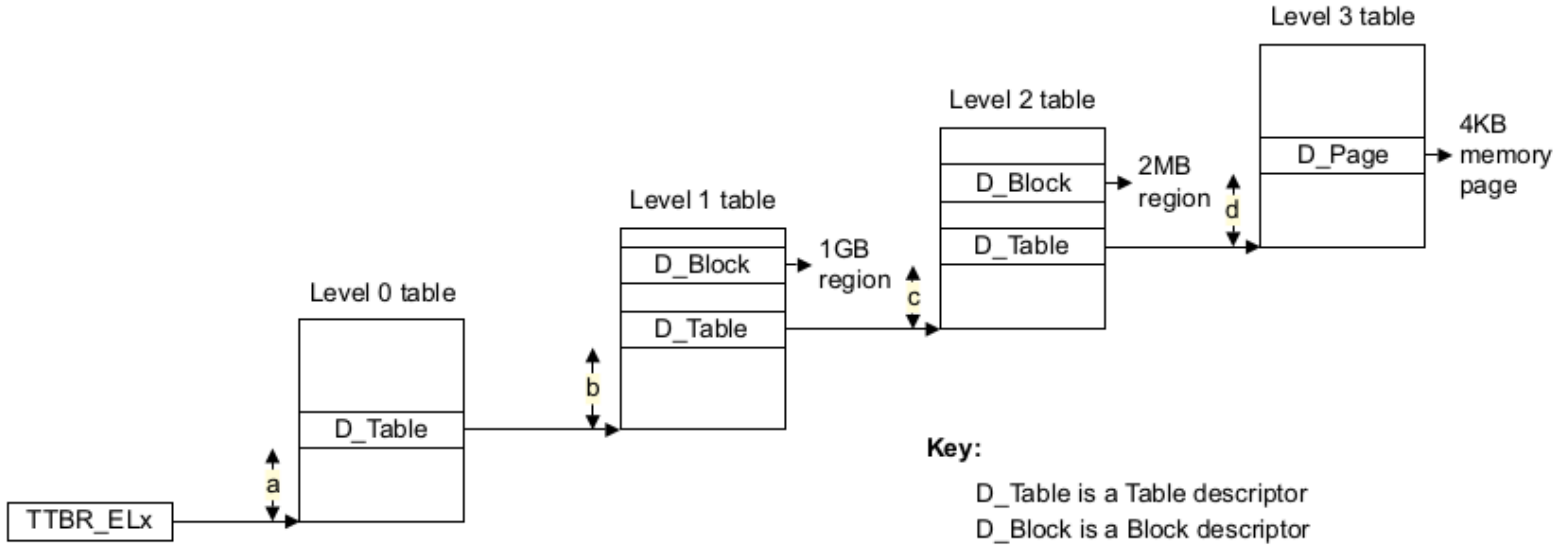
- Problem statement
- Introduction
  - ARM architecture & virtualization extensions
  - Samsung hypervisor
- Framework implementation & RKP analysis
  - System boot
    - **EL1 initialization**
  - Hypervisor boot
    - Hypervisor boot termination
  - Hypervisor initialization
    - Hypervisor exception handling
    - Hypervisor initialization
  - Demo
- Fuzzing
  - Dummy fuzzing
- Conclusions
- References

▷ EL1 initialization

- Setup not identical to Samsung Linux kernel
  - Framework is minimal in comparison
- Must not deviate much
  - Kernel – hypervisor dependency
  - Satisfy hypervisor requirements / assumptions
  - Ensure proper hypervisor interaction / behavior

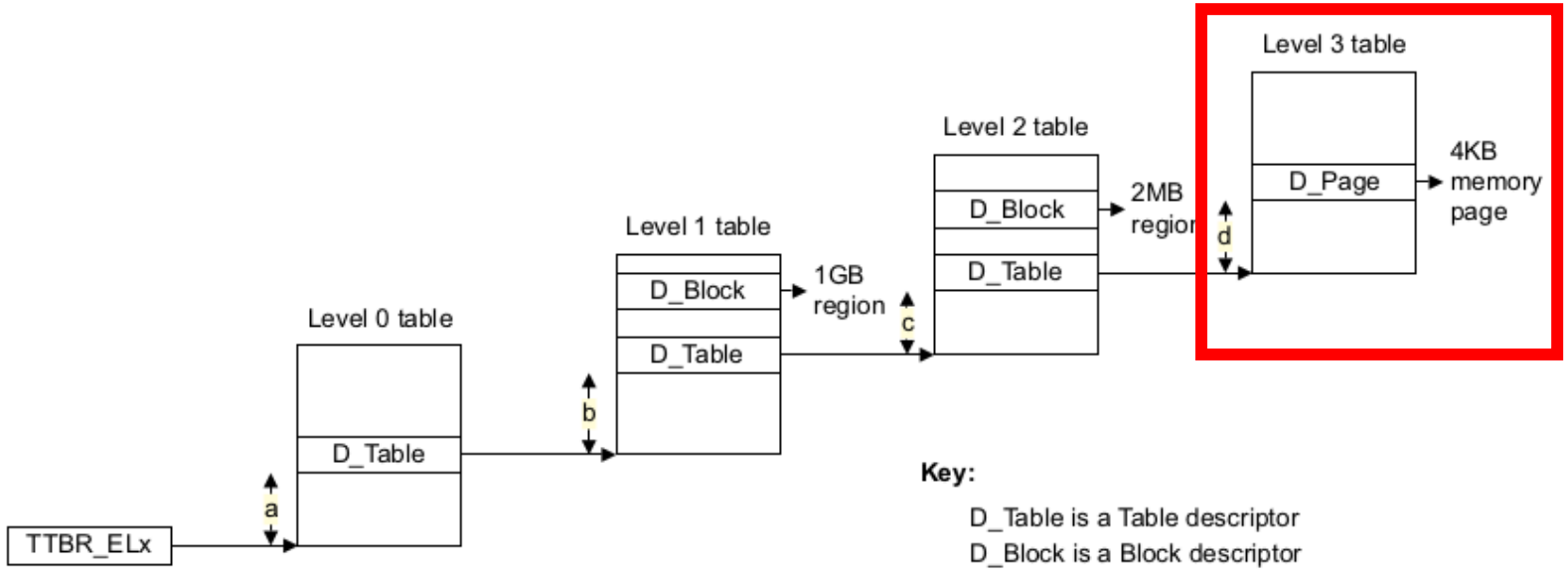
▷ EL1 initialization

- System registers
  - Value from source & oracles (discussed later)
  - Safely assume values as arbitrary for now
- Translation control (TCR_EL1) (for both TTBR1_EL1 and TTBR0_EL1)
  - 4kB Granule size
  - 512GB (39-bit) Input VA region
  - 1TB IPA
- Page table layout – entries
  - 4kB Granule (page) - 512 entries per page table
  - 512GB Input VA – Level 0 table not required



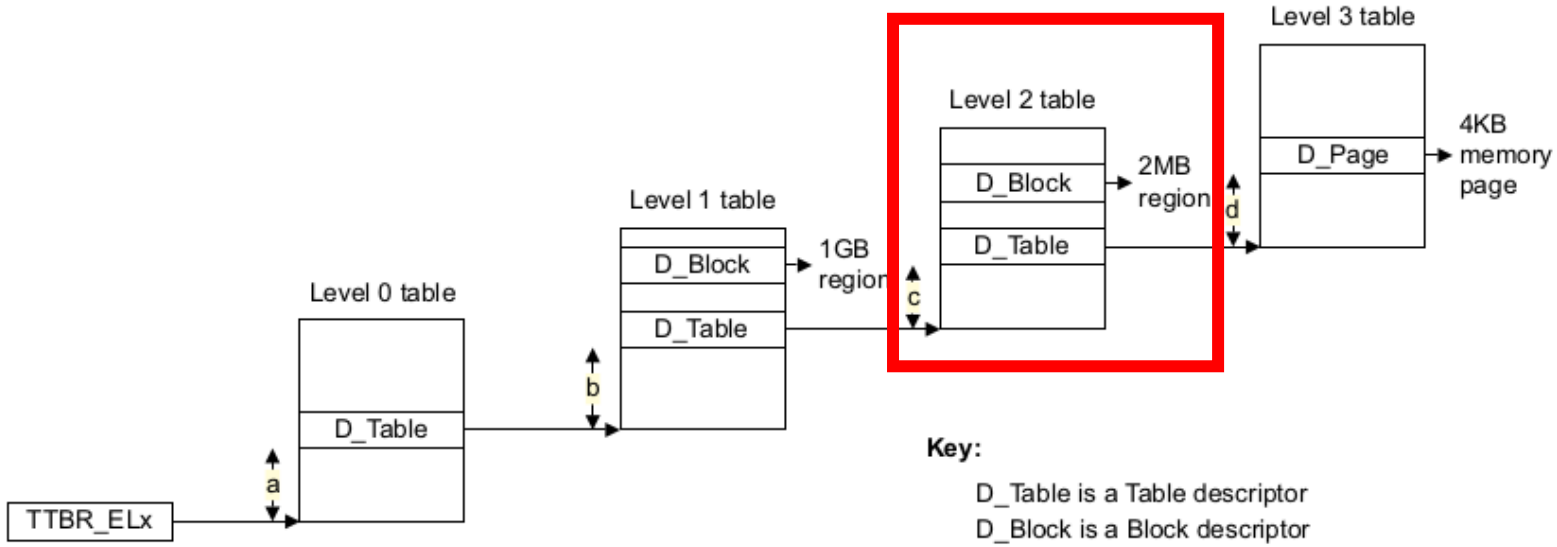
Virtual Address bits

[47:39]	[38:30]	[29:21]	[20:12]	[11:0]
Level 0 Table Index Each Entry can: - Point to L1 Table (No Block entries)	Level 1 Table Index Each Entry can: - Point to L2 Table - Point to a 1GB Block	Level 2 Table Index Each Entry can: - Point to L3 Table - Point to a 2MB Block	Level 3 Table Index Each Entry can: - Point to a 4KB Block (No Table entries)	Block Offset



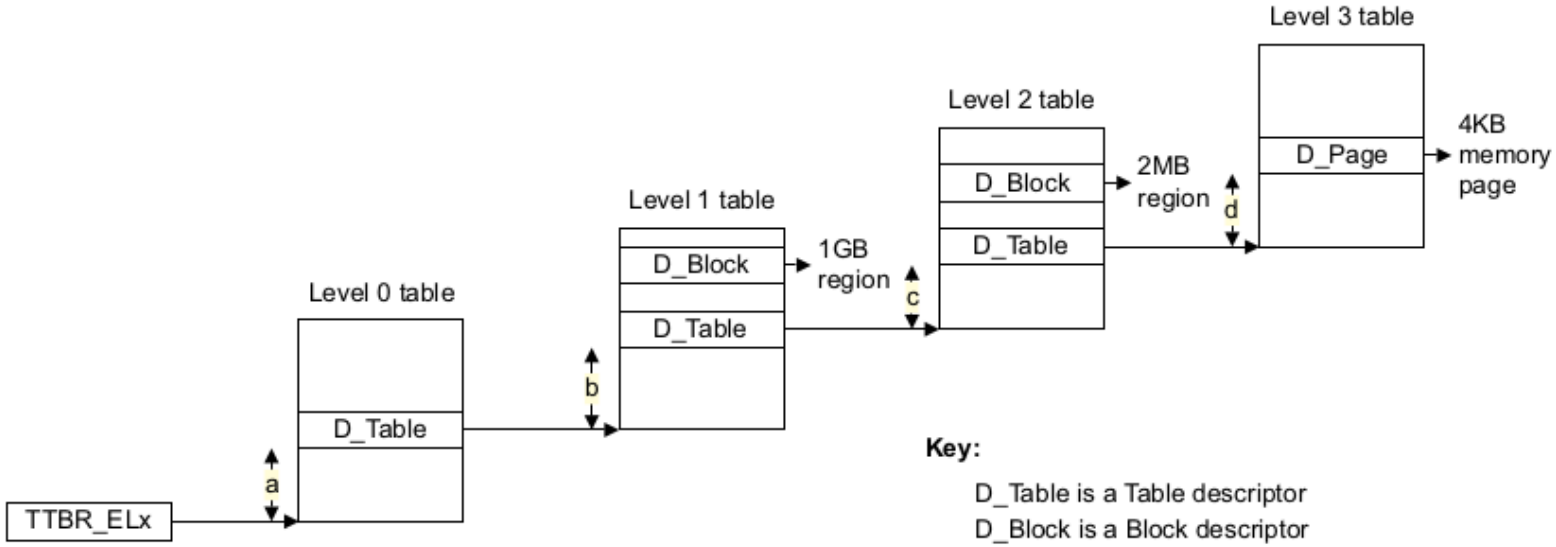
Virtual Address bits

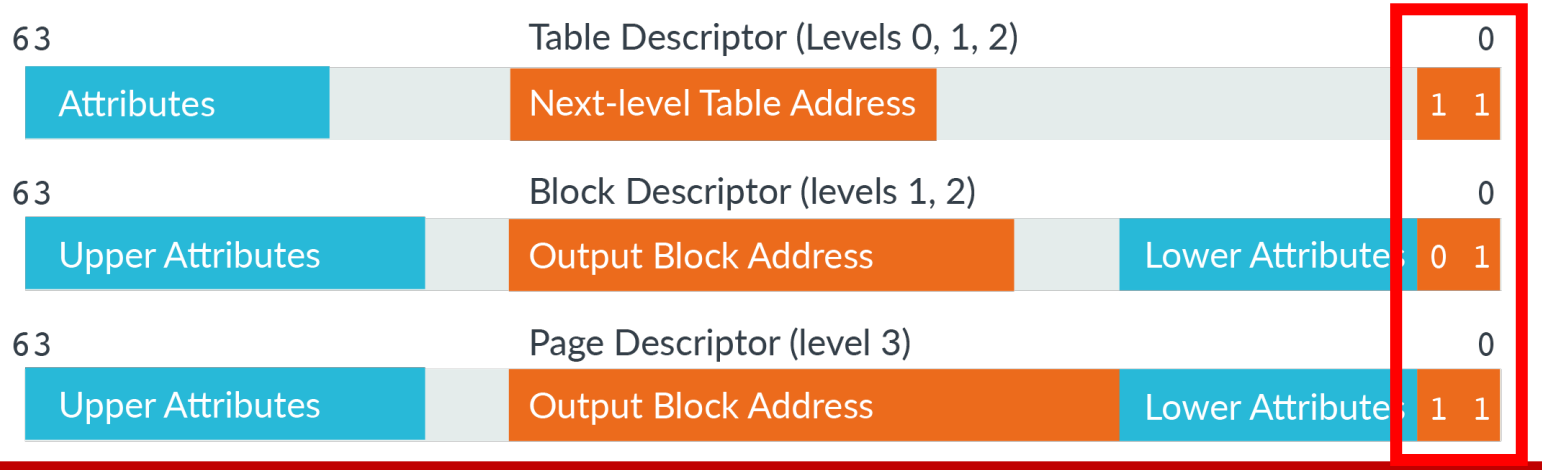
[47:39]	[38:30]	[29:21]	[20:12]	[11:0]
Level 0 Table Index Each Entry can: - Point to L1 Table (No Block entries)	Level 1 Table Index Each Entry can: - Point to L2 Table - Point to a 1GB Block	Level 2 Table Index Each Entry can: - Point to L3 Table - Point to a 2MB Block	Level 3 Table Index Each Entry can: - Point to a 4KB Block (No Table entries)	Block Offset

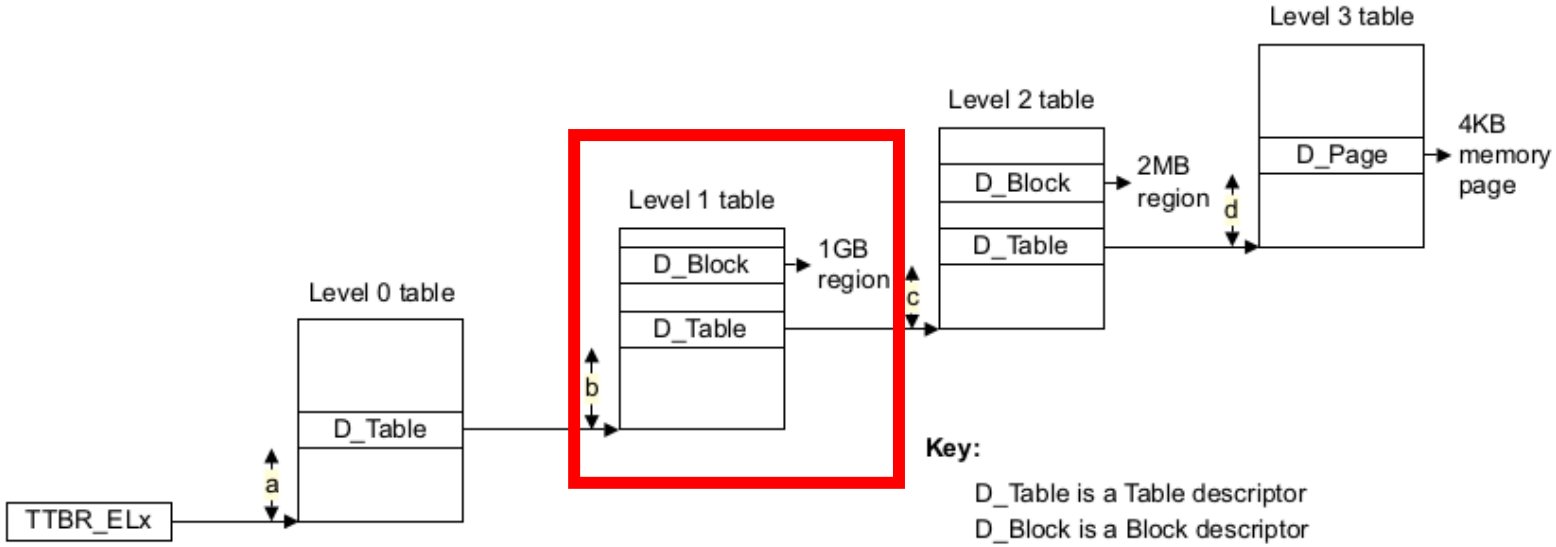


Virtual Address bits

[47:39]	[38:30]	[29:21]	[20:12]	[11:0]
Level 0 Table Index Each Entry can: - Point to L1 Table (No Block entries)	Level 1 Table Index Each Entry can: - Point to L2 Table - Point to a 1GB Block	Level 2 Table Index Each Entry can: - Point to L3 Table - Point to a 2MB Block	Level 3 Table Index Each Entry can: - Point to a 4KB Block (No Table entries)	Block Offset







Virtual Address bits

[47:39]	[38:30]	[29:21]	[20:12]	[11:0]
Level 0 Table Index Each Entry can: - Point to L1 Table (No Block entries)	Level 1 Table Index Each Entry can: - Point to L2 Table - Point to a 1GB Block	Level 2 Table Index Each Entry can: - Point to L3 Table - Point to a 2MB Block	Level 3 Table Index Each Entry can: - Point to a 4KB Block (No Table entries)	Block Offset

▷ EL1 initialization

- MMU enabling
- Two-page tables
  - Identity mapping TTBR0_EL1 (VA = PA)
  - TTBR1_EL1 (VA = PA + VA_OFFSET)
    - VA_OFFSET = 0xFFFFFF8000000000
- Create mapping using level 2 block entries (2MB)

▷ EL1 initialization

```
(gdb) disas
Dump of assembler code for function __enable_mmu:
   0x00000000800401a0 <+0>:     mov     x28, x30
   0x00000000800401a4 <+4>:     adrp    x25, 0x80089000 // TTBR1_EL1
   0x00000000800401a8 <+8>:     adrp    x26, 0x8008c000
   0x00000000800401ac <+12>:    bl      0x80040058 <__create_page_tables>
=> 0x00000000800401b0 <+16>:    isb
   0x00000000800401b4 <+20>:    mrs     x0, sctlr_el1
   0x00000000800401b8 <+24>:    orr     x0, x0, #0x1
End of assembler dump.

(gdb)  p/x ((0xffffff8000000000 + 0x80000000) >> 30) & 0x1ff
$19 = 0x2

(gdb)  x/gx ($TTBR1_EL1 + 2*8)
0x80089010:     0x000000008008a003

(gdb)  p/x ((0xffffff8000000000 + 0x80000000) >> 21) & 0x1ff
$20 = 0x0

(gdb)  x/gx 0x000000008008a000
0x8008a000:     0x0000000080000711

(gdb)  x/10i 0x0000000080000000
   0x80000000 <_reset>: ldr     x30, 0x80040000
   0x80000004 <_reset+4>:       mov     sp, x30
   0x80000008 <_reset+8>:       mrs     x0, currentel
```

▷ EL1 initialization

```
(gdb) disas
Dump of assembler code for function __enable_mmu:
   0x00000000800401a0 <+0>:     mov     x28, x30
   0x00000000800401a4 <+4>:     adrp    x25, 0x80089000 // TTBR1_EL1
   0x00000000800401a8 <+8>:     adrp    x26, 0x8008c000
   0x00000000800401ac <+12>:    bl      0x80040058 <__create_page_tables>
=> 0x00000000800401b0 <+16>:    isb
   0x00000000800401b4 <+20>:    mrs     x0, sctlr_el1
   0x00000000800401b8 <+24>:    orr     x0, x0, #0x1
End of assembler dump.

(gdb)  p/x ((0xffffff8000000000 + 0x80000000) >> 30) & 0x1ff
$19 = 0x2

(gdb)  x/gx ($TTBR1_EL1 + 2*8)
0x80089010:     0x000000008008a003

(gdb)  p/x ((0xffffff8000000000 + 0x80000000) >> 21) & 0x1ff
$20 = 0x0

(gdb)  x/gx 0x000000008008a000
0x8008a000:     0x0000000080000711

(gdb)  x/10i 0x0000000080000000
   0x80000000 <_reset>: ldr     x30, 0x80040000
   0x80000004 <_reset+4>:       mov     sp, x30
   0x80000008 <_reset+8>:       mrs     x0, currentel
```

▷ EL1 initialization

```
(gdb) disas
Dump of assembler code for function __enable_mmu:
   0x00000000800401a0 <+0>:     mov     x28, x30
   0x00000000800401a4 <+4>:     adrp    x25, 0x80089000 // TTBR1_EL1
   0x00000000800401a8 <+8>:     adrp    x26, 0x8008c000
   0x00000000800401ac <+12>:    bl      0x80040058 <__create_page_tables>
=> 0x00000000800401b0 <+16>:    isb
   0x00000000800401b4 <+20>:    mrs     x0, sctlr_el1
   0x00000000800401b8 <+24>:    orr     x0, x0, #0x1
End of assembler dump.

(gdb)  p/x ((0xffffff8000000000 + 0x80000000) >> 30) & 0x1ff
$19 = 0x2

(gdb)  x/gx ($TTBR1_EL1 + 2*8)
0x80089010:     0x000000008008a003

(gdb)  p/x ((0xffffff8000000000 + 0x80000000) >> 21) & 0x1ff
$20 = 0x0

(gdb)  x/gx 0x000000008008a000
0x8008a000:     0x0000000080000711

(gdb)  x/10i 0x0000000080000000
   0x80000000 <_reset>: ldr     x30, 0x80040000
   0x80000004 <_reset+4>:       mov     sp, x30
   0x80000008 <_reset+8>:       mrs     x0, currentel
```

▷ EL1 initialization

```
(gdb) disas
Dump of assembler code for function __enable_mmu:
   0x00000000800401a0 <+0>:     mov     x28, x30
   0x00000000800401a4 <+4>:     adrp    x25, 0x80089000 // TTBR1_EL1
   0x00000000800401a8 <+8>:     adrp    x26, 0x8008c000
   0x00000000800401ac <+12>:    bl      0x80040058 <__create_page_tables>
=> 0x00000000800401b0 <+16>:    isb
   0x00000000800401b4 <+20>:    mrs     x0, sctlr_el1
   0x00000000800401b8 <+24>:    orr     x0, x0, #0x1
End of assembler dump.

(gdb)  p/x ((0xffffff8000000000 + 0x80000000) >> 30) & 0x1ff
$19 = 0x2

(gdb)  x/gx ($TTBR1_EL1 + 2*8)
0x80089010:     0x000000008008a003

(gdb)  p/x ((0xffffff8000000000 + 0x80000000) >> 21) & 0x1ff
$20 = 0x0

(gdb)  x/gx 0x000000008008a000
0x8008a000:     0x0000000080000711

(gdb)  x/10i 0x0000000080000000
   0x80000000 <_reset>: ldr     x30, 0x80040000
   0x80000004 <_reset+4>:       mov     sp, x30
   0x80000008 <_reset+8>:       mrs     x0, currentel
```

▷ EL1 initialization

```
(gdb) disas
Dump of assembler code for function __enable_mmu:
   0x00000000800401a0 <+0>:     mov     x28, x30
   0x00000000800401a4 <+4>:     adrp    x25, 0x80089000 // TTBR1_EL1
   0x00000000800401a8 <+8>:     adrp    x26, 0x8008c000
   0x00000000800401ac <+12>:    bl      0x80040058 <__create_page_tables>
=> 0x00000000800401b0 <+16>:    isb
   0x00000000800401b4 <+20>:    mrs     x0, sctlr_el1
   0x00000000800401b8 <+24>:    orr     x0, x0, #0x1
End of assembler dump.

(gdb)  p/x ((0xffffff8000000000 + 0x80000000) >> 30) & 0x1ff
$19 = 0x2

(gdb)  x/gx ($TTBR1_EL1 + 2*8)
0x80089010:     0x000000008008a003

(gdb)  p/x ((0xffffff8000000000 + 0x80000000) >> 21) & 0x1ff
$20 = 0x0

(gdb)  x/gx 0x000000008008a000
0x8008a000:     0x0000000080000711

(gdb)  x/10i 0x0000000080000000
   0x80000000 <_reset>: ldr     x30, 0x80040000
   0x80000004 <_reset+4>:       mov     sp, x30
   0x80000008 <_reset+8>:       mrs     x0, currentel
```

▷ EL1 initialization

```
(gdb) disas
Dump of assembler code for function __enable_mmu:
   0x00000000800401a0 <+0>:     mov     x28, x30
   0x00000000800401a4 <+4>:     adrp    x25, 0x80089000 // TTBR1_EL1
   0x00000000800401a8 <+8>:     adrp    x26, 0x8008c000
   0x00000000800401ac <+12>:    bl      0x80040058 <__create_page_tables>
=> 0x00000000800401b0 <+16>:    isb
   0x00000000800401b4 <+20>:    mrs     x0, sctlr_el1
   0x00000000800401b8 <+24>:    orr     x0, x0, #0x1
End of assembler dump.

(gdb)  p/x ((0xffffff8000000000 + 0x80000000) >> 30) & 0x1ff
$19 = 0x2

(gdb)  x/gx ($TTBR1_EL1 + 2*8)
0x80089010:     0x000000008008a003

(gdb)  p/x ((0xffffff8000000000 + 0x80000000) >> 21) & 0x1ff
$20 = 0x0

(gdb)  x/gx 0x000000008008a000
0x8008a000:     0x0000000080000711

(gdb)  x/10i 0x0000000080000000
   0x80000000 <_reset>: ldr     x30, 0x80040000
   0x80000004 <_reset+4>:       mov     sp, x30
   0x80000008 <_reset+8>:       mrs     x0, currentel
```

▷ EL1 initialization

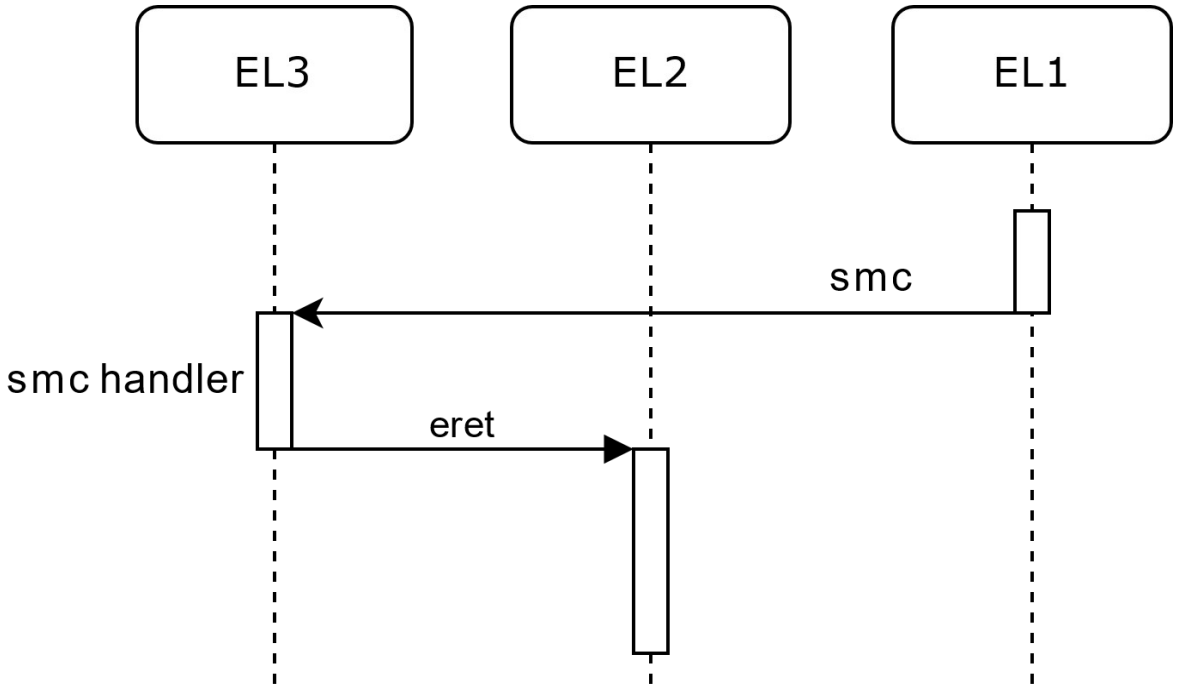
```
(gdb) disas
Dump of assembler code for function __enable_mmu:
   0x00000000800401a0 <+0>:     mov     x28, x30
   0x00000000800401a4 <+4>:     adrp    x25, 0x80089000 // TTBR1_EL1
   0x00000000800401a8 <+8>:     adrp    x26, 0x8008c000
   0x00000000800401ac <+12>:    bl      0x80040058 <__create_page_tables>
=> 0x00000000800401b0 <+16>:    isb
   0x00000000800401b4 <+20>:    mrs     x0, sctlr_el1
   0x00000000800401b8 <+24>:    orr     x0, x0, #0x1
End of assembler dump.

(gdb)  p/x ((0xffffff8000000000 + 0x80000000) >> 30) & 0x1ff
$19 = 0x2

(gdb)  x/gx ($TTBR1_EL1 + 2*8)
0x80089010:     0x000000008008a003

(gdb)  p/x ((0xffffff8000000000 + 0x80000000) >> 21) & 0x1ff
$20 = 0x0

(gdb)  x/gx 0x000000008008a000
0x8008a000:     0x0000000080000711

(gdb)  x/10i 0x0000000080000000
   0x80000000 <_reset>: ldr     x30, 0x80040000
   0x80000004 <_reset+4>:       mov     sp, x30
   0x80000008 <_reset+8>:       mrs     x0, currentel
```

▷ AGENDA

- Problem statement
- Introduction
  - ARM architecture & virtualization extensions
  - Samsung hypervisor
- Framework implementation & RKP analysis
  - System boot
    - EL1 initialization
  - **Hypervisor boot**
    - Hypervisor boot termination
  - Hypervisor initialization
    - Hypervisor exception handling
    - Hypervisor initialization
  - Demo
- Fuzzing
  - Dummy fuzzing
- Conclusions
- References

▷ Hypervisor boot

- RKP incorporated in framework image
  - Place in expected PA
    - ELF file - 0xB0100000
- hvc not available
  - EL1 -> EL3 -> EL2
- Drop to EL2 – fake state
  - ELR_EL3 – ELF entry point 0xB0101000
  - SPSR_EL3 – from EL2, AArch64
  - ESR_EL3 – irrelevant



▷ Hypervisor boot

- vmm_main()

```
int64_t vmm_main(int64_t hyp_base_arg, int64_t hyp_size_arg, char **stacks)
{
    ...

    memory_init();

    log_message("RKP_cdb5900c %sRKP_b826bc5a %s\n",
                            "Jul 11 2018", "11:19:43");

    /* various log messages and misc initializations */

    vmm_init();
    ...

    set_ttbr0_el2(&_static_s1_page_tables_start___ptr);
    s1_enable();

    set_vttbr_el2(&_static_s2_page_tables_start___ptr);
    s2_enable();

    ...
}
```

▷ Hypervisor boot

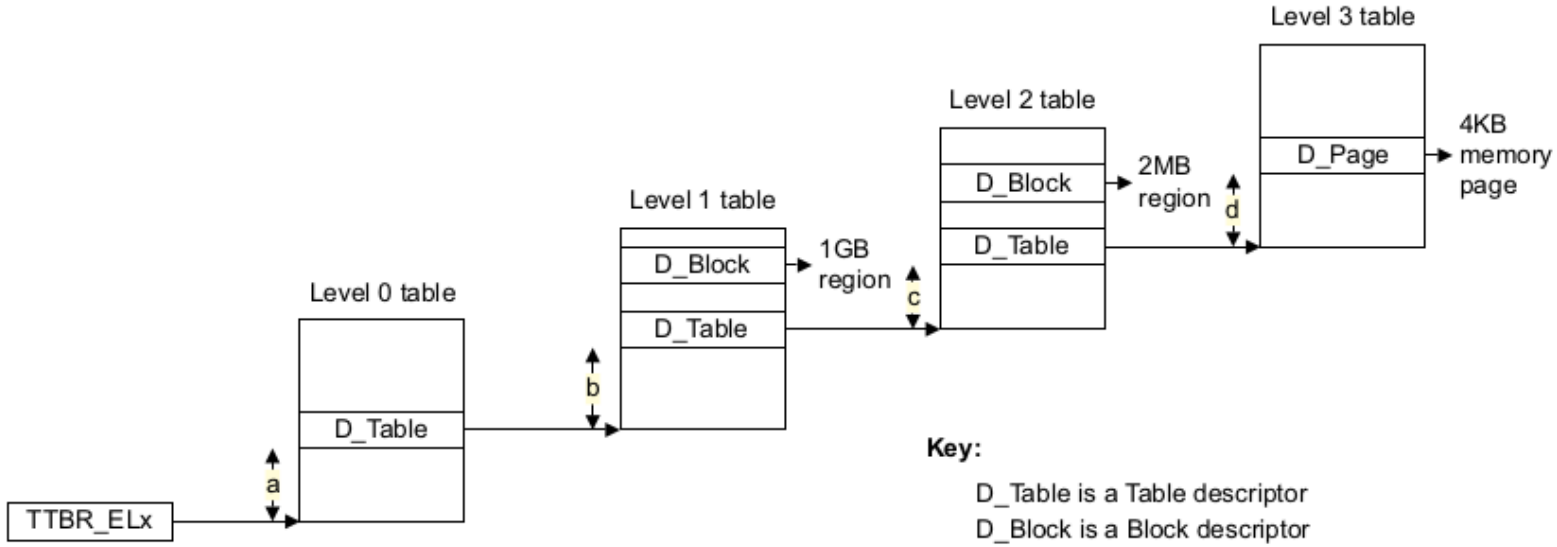
- memory_init()
  - Log buffer at 0xB0220000 (1st oracle)
  - Available via /proc/rkp_log

```
RKP_1f22e931 0xb0100000 RKP_dd15365a 40880   // file base: %p size %s
RKP_be7bb431 0xb0100000 RKP_dd15365a 100000  // region base: %p size %s
RKP_2db69dc3 0xb0220000 RKP_dd15365a 1f000   // memory log base: %p size %s
RKP_2c60d5a7 0xb0141000 RKP_dd15365a bf000   // heap base: %p size %s
```

- vmm_init()
  - Set VBAR_EL2
    - We can now invoke hvc from EL1!
      - Need to enable hvc command – SCR_EL3.HCE
  - Set HCR_EL2 values, dictate EL2-EL1 interaction
    - HCR_EL2.TVM, trap EL1 writes to system registers to EL2

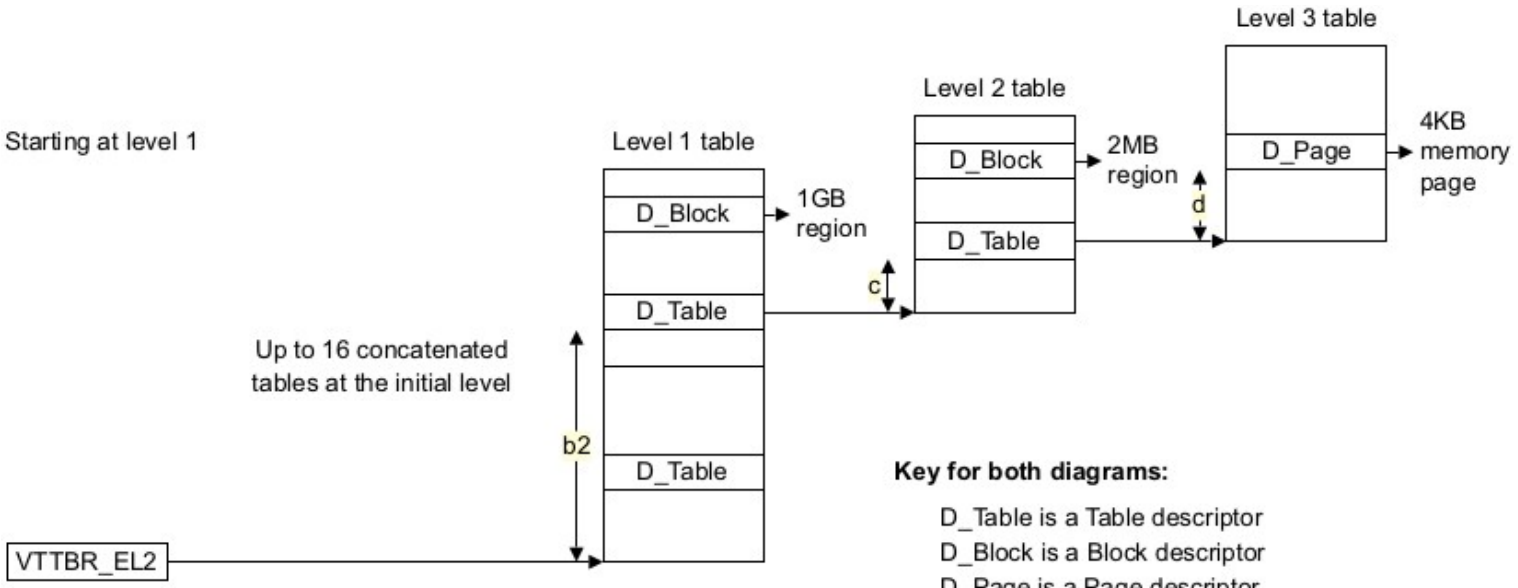
▷ Hypervisor boot

- s1_enable() – EL2 stage 1 translations
  - TTBR0_EL2 – static page tables
  - TCR_EL2
    - 4kB Granule size
    - 1TB Input VA
    - 1TB PA space
- s2_enable() – EL2 stage 2 translations
  - VTTBR_EL2 – static page tables
  - VTCR_EL2
    - 4kB Granule size
    - 1TB (40-bit) Input VA
    - 1TB PA space
    - Start at Level 1 with concatenated tables



Virtual Address bits

[47:39]	[38:30]	[29:21]	[20:12]	[11:0]
Level 0 Table Index Each Entry can: - Point to L1 Table (No Block entries)	Level 1 Table Index Each Entry can: - Point to L2 Table - Point to a 1GB Block	Level 2 Table Index Each Entry can: - Point to L3 Table - Point to a 2MB Block	Level 3 Table Index Each Entry can: - Point to a 4KB Block (No Table entries)	Block Offset



Key for both diagrams:

D_Table is a Table descriptor
D_Block is a Block descriptor
D_Page is a Page descriptor

a Indexed by IA[*n*:39], where IA width is (*n*+1) bits
b1 Indexed by IA[38:30]
b2 Indexed by IA[*n*:30], where IA width is (*n*+1) bits
c Indexed by IA[29:21]
d Indexed by IA[20:12]

Virtual Address bits

[47:39]	39 [~~38~~:30]	[29:21]	[20:12]	[11:0]
Level 0 Table Index Each Entry can: - Point to L1 Table (No Block entries	Level 1 Table Index Each Entry can: - Point to L2 Table - Point to a 1GB Block	Level 2 Table Index Each Entry can: - Point to L3 Table - Point to a 2MB Block	Level 3 Table Index Each Entry can: - Point to a 4KB Block (No Table entries)	Block Offset

▷ Hypervisor boot

- Tool to dump stage 2 table and attributes

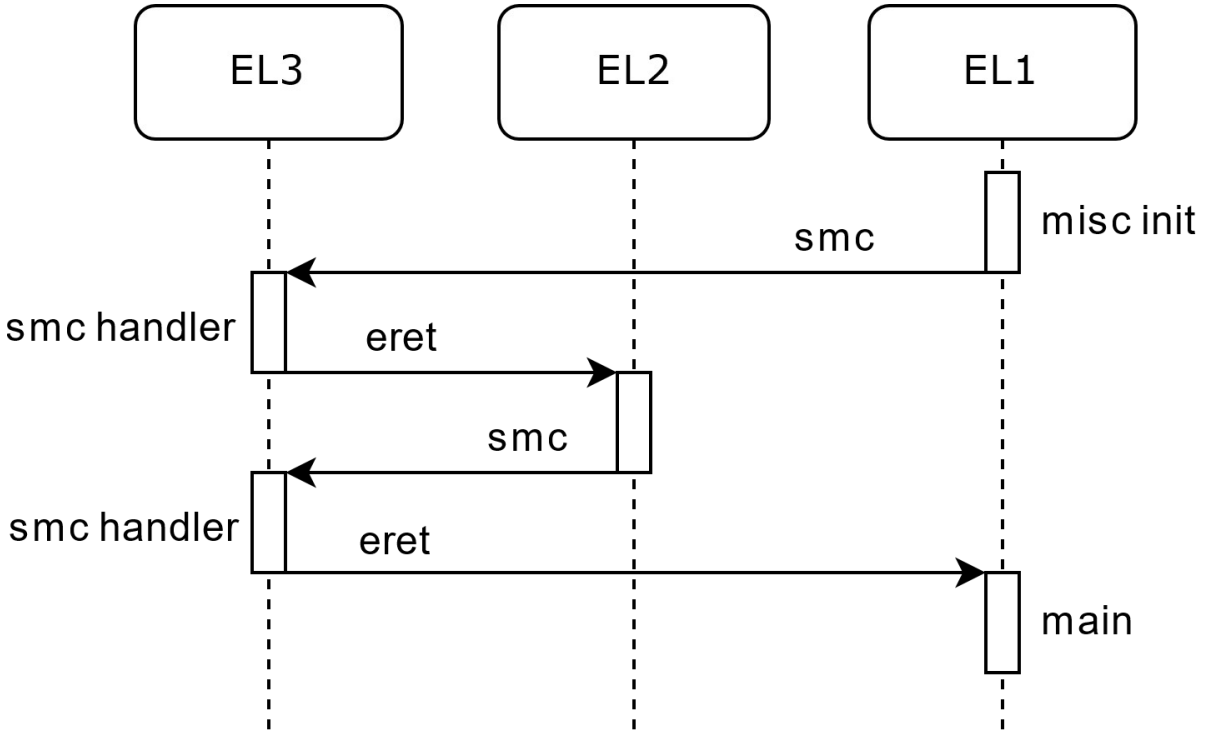
```
(gdb) pagewalk

################################################
#     Dump Second Stage Translation Tables     #
################################################
PA Size: 40-bits
Starting Level: 1
IPA range: 0x000000fffffffff
Page Size: 4KB

...
Third level: 0x1c07d000-0x1c07e000: S2AP=11, XN=10
Third level: 0x1c07e000-0x1c07f000: S2AP=11, XN=10
...
second level block: 0xbfc00000-0xbfe00000: S2AP=11, XN=0
second level block: 0xbfe00000-0xc0000000: S2AP=11, XN=0
first level block: 0xc0000000-0x100000000: S2AP=11, XN=0
first level block: 0x880000000-0x8c0000000: S2AP=11, XN=0
...
```

▷ AGENDA

- Problem statement
- Introduction
  - ARM architecture & virtualization extensions
  - Samsung hypervisor
- Framework implementation & RKP analysis
  - System boot
    - EL1 initialization
  - Hypervisor boot
    - **Hypervisor boot termination**
  - Hypervisor initialization
    - Hypervisor exception handling
    - Hypervisor initialization
  - Demo
- Fuzzing
  - Dummy fuzzing
- Conclusions
- References

▷ Hypervisor boot termination

- Terminate via smc – EL2 -> EL3
  - X0 special value 0xC2000401
  - X1, initialization status
- EL3 new exception – restore state and return to EL1
- No need to preserve state
  - Reset stack pointers
  - Drop to EL1 function



▷ Hypervisor boot termination

- System constraint – framework .text at 0x80000000
- EL1 return address must be mapped in Stage 2 translation tables

```
(gdb) pagewalk

##################################################
#     Dump Second Stage Translation Tables       #
##################################################
...
Third level: 0x1c07e000-0x1c07f000: S2AP=11, XN=10
Third level: 0x1c07f000-0x1c080000: S2AP=11, XN=10
Third level: 0x80000000-0x80001000: S2AP=1, XN=0
Third level: 0x80001000-0x80002000: S2AP=1, XN=0
...
```

▷ AGENDA

- Problem statement
- Introduction
  - ARM architecture & virtualization extensions
  - Samsung hypervisor
- Framework implementation & RKP analysis
  - System boot
    - EL1 initialization
  - Hypervisor boot
    - Hypervisor boot termination
  - Hypervisor initialization
    - **Hypervisor exception handling**
    - Hypervisor initialization
  - Demo
- Fuzzing
  - Dummy fuzzing
- Conclusions
- References

▷ Hypervisor exception handling

- All exceptions lead to vmm_dispatch()
  - Only synchronous exceptions

```
stp     X1, X0, [SP,#exception_frame]!
...
mov     X0, #0x400          // Lower AArch64
mov     X1, #0              // Synchronous Exception
mov     X2, SP              // Exception frame, holding args from EL1

bl      vmm_dispatch
...
ldp     X1, X0, [SP+0x10+exception_frame],#0x10
clrex
eret
```

▷ Hypervisor exception handling

- vmm_synchronous_handler()
  - ESR_EL2 – get reason for exception
  - hvc invocations handled by rkp_main()

```
int64_t vmm_synchronous_handler(int64_t from_el_offset,
                int64_t exception_type, exception_frame *exception_frame) {

    esr_el2 = get_esr_el2();
    ...

    switch ( esr_el2 >> 26 )   /* Exception Class */
    {
        case 0x12:        /* HVC from AArch32 */
        case 0x16:        /* HVC from AArch64 */


            if ((exception_frame->x0 & 0xFFF00000) == 0x83800000)
                rkp_main(exception_frame->x0, exception_frame);
            ...
            return 0;
    ...
}
```

▷ Hypervisor exception handling

- hvc first argument X0 – command
  - Command prefix 0x83800000

```
if ((exception_frame->x0 & 0xFFF00000) == 0x83800000)
    rkp_main(exception_frame->x0, exception_frame);
```

- Command id – function identifier
  - Shifted by 12
  - OR'ed with prefix

```
void rkp_main(unsigned int64_t command, exception_frame *exception_frame)
{
    hvc_cmd = (command >> 12) & 0xFF;

    ...

    my_check_hvc_command(hvc_cmd);
    switch ( hvc_cmd )
    ...
}
```

▷ Hypervisor exception handling

- check_hvc_command()
  - Command id smaller than 0x9F
  - Check command counter
    - Some commands must be called less than `counter` times

Function	ID	Command	Counter
rkp_init	0x0	0x83800000	0
rkp_def_init	0x1	0x83801000	1
rkp_pgd_set	0x21	0x83821000	-1
rkp_pmd_set	0x22	0x83822000	-1

▷ AGENDA

- Problem statement
- Introduction
  - ARM architecture & virtualization extensions
  - Samsung hypervisor
- Framework implementation & RKP analysis
  - System boot
    - EL1 initialization
  - Hypervisor boot
    - Hypervisor boot termination
  - Hypervisor initialization
    - Hypervisor exception handling
    - **Hypervisor initialization**
  - Demo
- Fuzzing
  - Dummy fuzzing
- Conclusions
- References

▷ Hypervisor initialization

- rkp_init() - command id 0 (0x83800000)
- Expects struct with info about kernel
  - Magic value
  - vmalloc range
  - Page table addresses
  - .text, .rodata regions
  - Bitmaps (Samsung specific)
  - Other Samsung specific regions
  - Etc.

▷ Hypervisor initialization

```
void rkp_init(exception_frame *exception_frame)
{
    ...

    rkp_debug_log_init();
    ...

    if ( rkp_init_values->magic - 0x5AFE0001 <= 1 ){

        ...

        /* misc initializations and debug logs */

        rkp_debug_log("RKP_6398d0cb", hcr_el2,
                      sctlr_el2, rkp_init_values->magic);

        /* more debug logs */

        if ( rkp_paging_init() )
        {
            ...

            my_initialize_hvc_cmd_counter();

            ...
        }
        ...
    }
    ...
}
```

▷ Hypervisor initialization

- rkp_debug_log_init()
  - Buffer at 0xB0200000 (2nd oracle)
  - Available also via /proc/rkp_log

```
/* misc initializations and debug logs */

rkp_debug_log("RKP_6398d0cb", hcr_el2, sctlr_el2, rkp_init_values->magic);

/* more debug logs */
```

```
0000000000000000    neoswbuilder-DeskTop RKP64_01aa4702
0000000000000000    Jul 11 2018
0000000000000000    11:19:42

/* hcr_el2 */      /* sctlr_el2 */
  84000003          30cd1835           5afe0001  RKP_6398d0cb

/* tcr_el2 */      /* tcr_el1 */
  80823518          32b5593519         5afe0001  RKP_64996474

/* mair_el2 */     /* mair_el1 */
21432b2f914000ff  0000bbff440c0400     5afe0001  RKP_bd1f621f
```

▷ Hypervisor initialization

- Two modes – Magic Value
  - Once set cannot change
- Normal mode – 0x5AFE0001
- Test mode – 0x5AFE0002
  - Enable testing
  - Disable command counters
  - Additional test functions

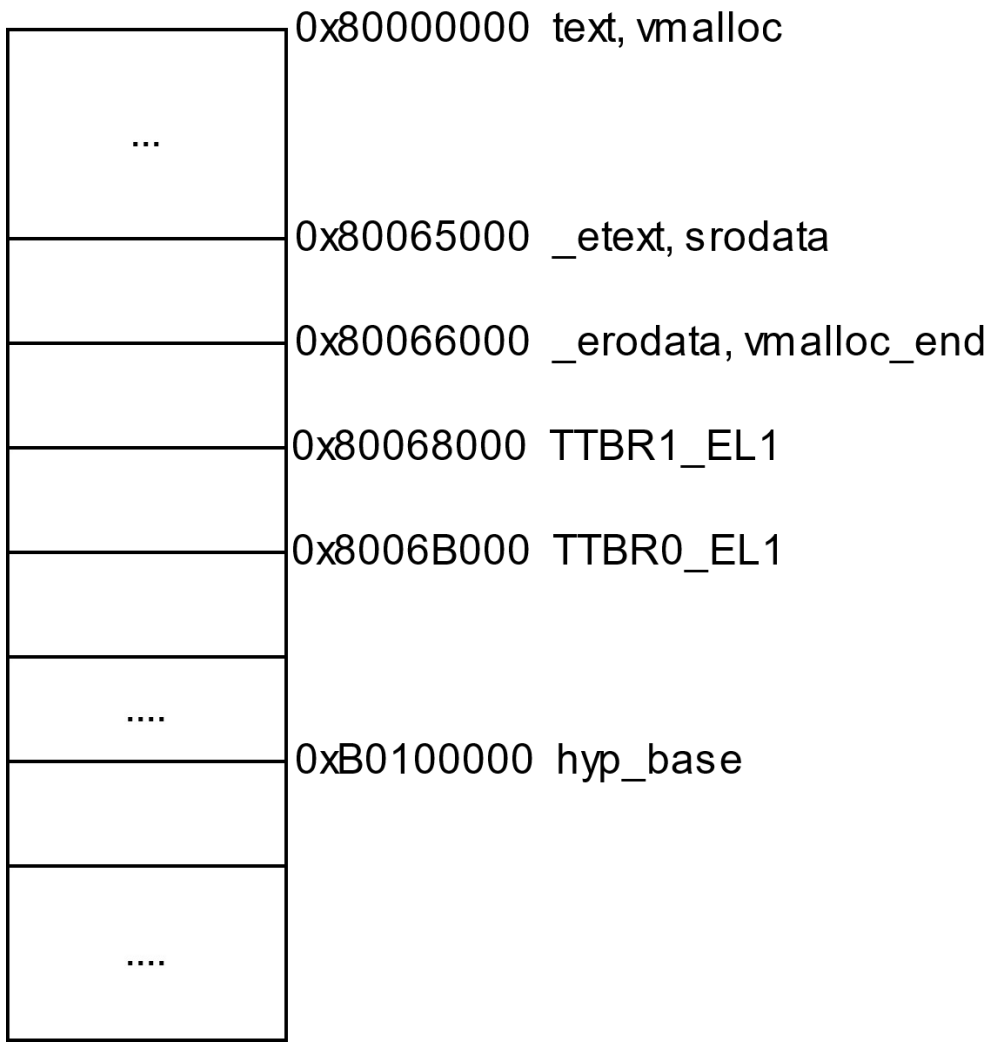
```
void rkp_init(exception_frame *exception_frame)
{
    ...

    rkp_debug_log_init();
    ...

    if ( rkp_init_values->magic - 0x5AFE0001 <= 1 ){

        if ( rkp_init_values->magic == 0x5AFE0002 )
        {
            /* enable test mode */
        }

    ...
}
```

▷ Hypervisor initialization

- rkp_paging_init() - most important function
- Checks regarding memory layout
  - Framework EL1 layout must satisfy all checks
- Change memory region attributes for stage 2 translations
- Set internal information for memory region housekeeping
- Unmap sensitive regions from stage 2 tables
- If anything fails, RKP does not initialize

▷ Hypervisor initialization

- Sample layout with essential regions
- Some entries are not implemented but regions required either way
  - e.g. vmalloc
- System constraint 3GB RAM



▷ Hypervisor initialization

- rkp_def_init() - command id 1 (0x83801000)
- Kernel text region: read only
- TTBR1_EL1: read only and not executable

```
void rkp_def_init(void)
{
    ...

    rkp_s2_change_range_permission(text_pa, etext_pa,
                                   0x80, 1, 1));
    ...

    rkp_l1pgt_process_table(swapper_pg_dir, 1, 1);
    ...
}
```

```
// EL1 text before rkp_s2_change_range_permission()
Third level: 0x80000000-0x80001000: S2AP=11, XN=0

// EL1 text after rkp_s2_change_range_permission()
Third level: 0x80000000-0x80001000: S2AP=1, XN=0


// TTBR1_EL1 before rkp_l1pgt_process_table()
Third level: 0x80088000-0x80089000: S2AP=11, XN=0
Third level: 0x80089000-0x8008a000: S2AP=11, XN=0

// TTBR1_EL1 after rkp_l1pgt_process_table()
Third level: 0x80088000-0x80089000: S2AP=1, XN=10
Third level: 0x80089000-0x8008a000: S2AP=1, XN=10
```

▷ Hypervisor initialization

- Can now interact with RKP!
- Did not follow original kernel initialization
  - More init routines
- Introduced EL2 basic concepts
- Can start investigating fuzzing setups!

- Must be aware of hypervisor smc invocations
  - Handle them in EL3
  - Must also consider while fuzzing
- One available in RKP for verification of Samsung specific images
  - Available via specific hvc command
  - Can safely skip for now

▷ AGENDA

- Problem statement
- Introduction
  - ARM architecture & virtualization extensions
  - Samsung hypervisor
- Framework implementation & RKP analysis
  - System boot
    - EL1 initialization
  - Hypervisor boot
    - Hypervisor boot termination
  - Hypervisor initialization
    - Hypervisor exception handling
    - Hypervisor initialization
  - **Demo**
- Fuzzing
  - Dummy fuzzing
- Conclusions
- References

▷ Demo

▷ AGENDA

- Problem statement
- Introduction
  - ARM architecture & virtualization extensions
  - Samsung hypervisor
- Framework implementation & RKP analysis
  - System boot
    - EL1 initialization
  - Hypervisor boot
    - Hypervisor boot termination
  - Hypervisor initialization
    - Hypervisor exception handling
    - Hypervisor initialization
  - Demo
- Fuzzing
  - **Dummy fuzzing**
- Conclusions
- References

▷ Dummy Fuzzing

- Numerous approaches depending on requirements
- ARM semihosting – Communicate with host and use its I/O facilities
  - Pros
    - Minimal QEMU modification
  - Cons
    - Need to implement sync with host
    - Slower (probably)
- Directly from QEMU
  - Pros
    - Sync by design
    - Faster
  - Cons
    - QEMU modification required

▷ Dummy Fuzzing

- QEMU creates test – delivers to guest VM
- Utilize brk instruction
- Pass commands via ESR_ELx.ISS
  - brk #imm (ISS = #imm)
- Catch brk in QEMU
- Can create arbitrary fuzzing harnesses
  - Sync QEMU and framework

▷ Dummy Fuzzing

- Handling brk in QEMU

```
/* Handle a CPU exception for A and R profile CPUs.
   ...
 */
void arm_cpu_do_interrupt(CPUState *cs)
{
    ...

    // Handle the break instruction
    if (cs->exception_index == EXCP_BKPT) {

        handle_brk(cs, env);

        ...
    }
    ...
}
```

▷ Dummy Fuzzing

- Sample harness – fuzz hvc commands
  - Invoke brk to create testcase
    - Random byte for command id -> X0 (1st argument)
    - Copy page with random values to guest buffer -> X1 (2nd argument)
      - Required by many RKP functions (RKP specific)
  - Invoke hvc to fuzz with testcase
  - Repeat
  - Profit!

▷ Dummy Fuzzing

QEMU

```
switch (syndrome) {
    ...

    case 1: ; // dummy fuzz
        uint8_t cmd = random() & 0xFF;


        /*
         * Write host buffer buf to guest buffer pointed to
         * by register X0 during brk invocation
         */
        cpu_memory_rw_debug(cs, env->xregs[0], buf, l, 1) < 0;

        fuzz_cpu_state.xregs[0] = 0x83800000 | (cmd << 12);
        fuzz_cpu_state.xregs[1] = env->xregs[0];


        env->xregs[0] = fuzz_cpu_state.xregs[0];
        env->xregs[1] = fuzz_cpu_state.xregs[1];
        break;
```

Framework

```
void el1_main(void) {
    framework_rkp_init();
    rkp_call(RKP_DEF_INIT, 0, 0, 0, 0, 0);

    for(; ;){              // fuzzing loop
        __break_fuzz();    // create fuzzed values
        rkp_call_fuzz(); // invoke RKP
    }
}

__break_fuzz:
    ldr x0, =rand_buf
    brk #1
    ret
ENDPROC(__break_fuzz)

rkp_call_fuzz:
    hvc #0
    ret
ENDPROC(rkp_call_fuzz)
```

▷ Dummy Fuzzing

QEMU

```
switch (syndrome) {
    ...

case 1: ; // dummy fuzz
        uint8_t cmd = random() & 0xFF;


        /*
         * Write host buffer buf to guest buffer pointed to
         * by register X0 during brk invocation
         */
        cpu_memory_rw_debug(cs, env->xregs[0], buf, l, 1) < 0;

        fuzz_cpu_state.xregs[0] = 0x83800000 | (cmd << 12);
        fuzz_cpu_state.xregs[1] = env->xregs[0];


        env->xregs[0] = fuzz_cpu_state.xregs[0];
        env->xregs[1] = fuzz_cpu_state.xregs[1];
        break;
```

Framework

```
void el1_main(void) {
    framework_rkp_init();
    rkp_call(RKP_DEF_INIT, 0, 0, 0, 0, 0);

    for(;;){                    // fuzzing loop
        __break_fuzz();         // create fuzzed values
        rkp_call_fuzz();        // invoke RKP
    }
}

__break_fuzz:
    ldr x0, =rand_buf
    brk #1
    ret
ENDPROC(__break_fuzz)

rkp_call_fuzz:
    hvc #0
    ret
ENDPROC(rkp_call_fuzz)
```

▷ Dummy Fuzzing

QEMU

```
switch (syndrome) {
    ...

    case 1: ; // dummy fuzz
        uint8_t cmd = random() & 0xFF;


        /*
         * Write host buffer buf to guest buffer pointed to
         * by register X0 during brk invocation
         */
        cpu_memory_rw_debug(cs, env->xregs[0], buf, l, 1) < 0;

        fuzz_cpu_state.xregs[0] = 0x83800000 | (cmd << 12);
        fuzz_cpu_state.xregs[1] = env->xregs[0];


        env->xregs[0] = fuzz_cpu_state.xregs[0];
        env->xregs[1] = fuzz_cpu_state.xregs[1];
        break;
```

Framework

```
void el1_main(void) {
    framework_rkp_init();
    rkp_call(RKP_DEF_INIT, 0, 0, 0, 0, 0);

    for(; ;){              // fuzzing loop
        __break_fuzz();    // create fuzzed values
        rkp_call_fuzz(); // invoke RKP
    }
}

__break_fuzz:
    ldr x0, =rand_buf
    brk #1
    ret
ENDPROC(__break_fuzz)

rkp_call_fuzz:
    hvc #0
    ret
ENDPROC(rkp_call_fuzz)
```

▷ Dummy Fuzzing

QEMU

```
switch (syndrome) {
    ...

    case 1: ; // dummy fuzz
        uint8_t cmd = random() & 0xFF;

        /*
         * Write host buffer buf to guest buffer pointed to
         * by register X0 during brk invocation
         */
        cpu_memory_rw_debug(cs, env->xregs[0], buf, l, 1) < 0;

        fuzz_cpu_state.xregs[0] = 0x83800000 | (cmd << 12);
        fuzz_cpu_state.xregs[1] = env->xregs[0];


        env->xregs[0] = fuzz_cpu_state.xregs[0];
        env->xregs[1] = fuzz_cpu_state.xregs[1];
        break;
```

Framework

```
void el1_main(void) {
    framework_rkp_init();
    rkp_call(RKP_DEF_INIT, 0, 0, 0, 0, 0);

    for(; ;){              // fuzzing loop
        __break_fuzz();    // create fuzzed values
        rkp_call_fuzz(); // invoke RKP
    }
}

__break_fuzz:
    ldr x0, =rand_buf
    brk #1
    ret
ENDPROC(__break_fuzz)

rkp_call_fuzz:
    hvc #0
    ret
ENDPROC(rkp_call_fuzz)
```

▷ Dummy Fuzzing

QEMU

```
switch (syndrome) {
    ...

    case 1: ; // dummy fuzz
        uint8_t cmd = random() & 0xFF;


        /*
         * Write host buffer buf to guest buffer pointed to
         * by register X0 during brk invocation
         */
        cpu_memory_rw_debug(cs, env->xregs[0], buf, l, 1) < 0;

        fuzz_cpu_state.xregs[0] = 0x83800000 | (cmd << 12);
        fuzz_cpu_state.xregs[1] = env->xregs[0];


        env->xregs[0] = fuzz_cpu_state.xregs[0];
        env->xregs[1] = fuzz_cpu_state.xregs[1];
        break;
```

Framework

```
void el1_main(void) {
    framework_rkp_init();
    rkp_call(RKP_DEF_INIT, 0, 0, 0, 0, 0);

    for(; ;){                   // fuzzing loop
        break_fuzz();           // create fuzzed values
        rkp_call_fuzz();        // invoke RKP
    }
}

__break_fuzz:
    ldr x0, =rand_buf
    brk #1
    ret
ENDPROC(__break_fuzz)

rkp_call_fuzz:
    hvc #0
    ret
ENDPROC(rkp_call_fuzz)
```

▷ Dummy Fuzzing

- Nothing happens... WTH!!?

▷ Dummy Fuzzing

- What is a crash?
- Bare metal – nothing to "crash"
- Abort exceptions
- Handle them in QEMU
  - Create crash log
  - Reset QEMU / VM

```
void arm_cpu_do_interrupt(CPUState *cs)
{
    ...

    // Handle the instruction or data abort
    if (cs->exception_index == EXCP_PREFETCH_ABORT ||
          cs->exception_index == EXCP_DATA_ABORT ) {

        handle_abort(cs, env);

        ...
    }
    ...
}
```

▷ Dummy Fuzzing

```
********** Data\Instruction abort! **********
FAR = <redacted>          ELR = <redacted>
Fuzz x0 = <redacted>      Fuzz x1 = <redacted>

********** CPU State **********
 PC= <redacted>       X00= <redacted>       X01=000000000000
X02=00000000800c5000 X03=0000000000000000 X04=000000000000
....
X29= <redacted>       X30= <redacted>        SP = <redacted>
PSTATE=600003c9 -ZC- NS EL2h

********** Disassembly **********
<redacted>


********** Memory Dump **********
...

X02: 0x00000000800c5000
...
00000000800c4fe0: 0x0000000000000000 0x0000000000000000
00000000800c4ff0: 0x0000000000000000 0x0000000000000000
00000000800c5000: 0x21969a71a5b30938 0xc6d843c68f2f38be
00000000800c5010: 0xd7a1a2d7948ffd7e 0x42793a9f98647619
00000000800c5020: 0x87c01b08bb98d031 0x1949658c38220d4d

...

********** End of report **********
```

▷ Dummy Fuzzing

- Special cases
  - Result in endless loop
  - Identify in QEMU via address
    - Not scaling well
- vmm_panic()
  - Only few places
  - Action: reset system
- rkp_policy_violation()
  - Assert() logic
  - Action: reset system

▷ AGENDA

- Problem statement
- Introduction
  - ARM architecture & virtualization extensions
  - Samsung hypervisor
- Framework implementation & RKP analysis
  - System boot
    - EL1 initialization
  - Hypervisor boot
    - Hypervisor boot termination
  - Hypervisor initialization
    - Hypervisor exception handling
    - Hypervisor initialization
  - Demo
- Fuzzing
  - Dummy fuzzing
- **Conclusions**
- References

▷ Conclusions

- Introduced key concepts and discussed the various considerations required to emulate hypervisors
- Discussed basic approaches to fuzzing
- Build your own frameworks!

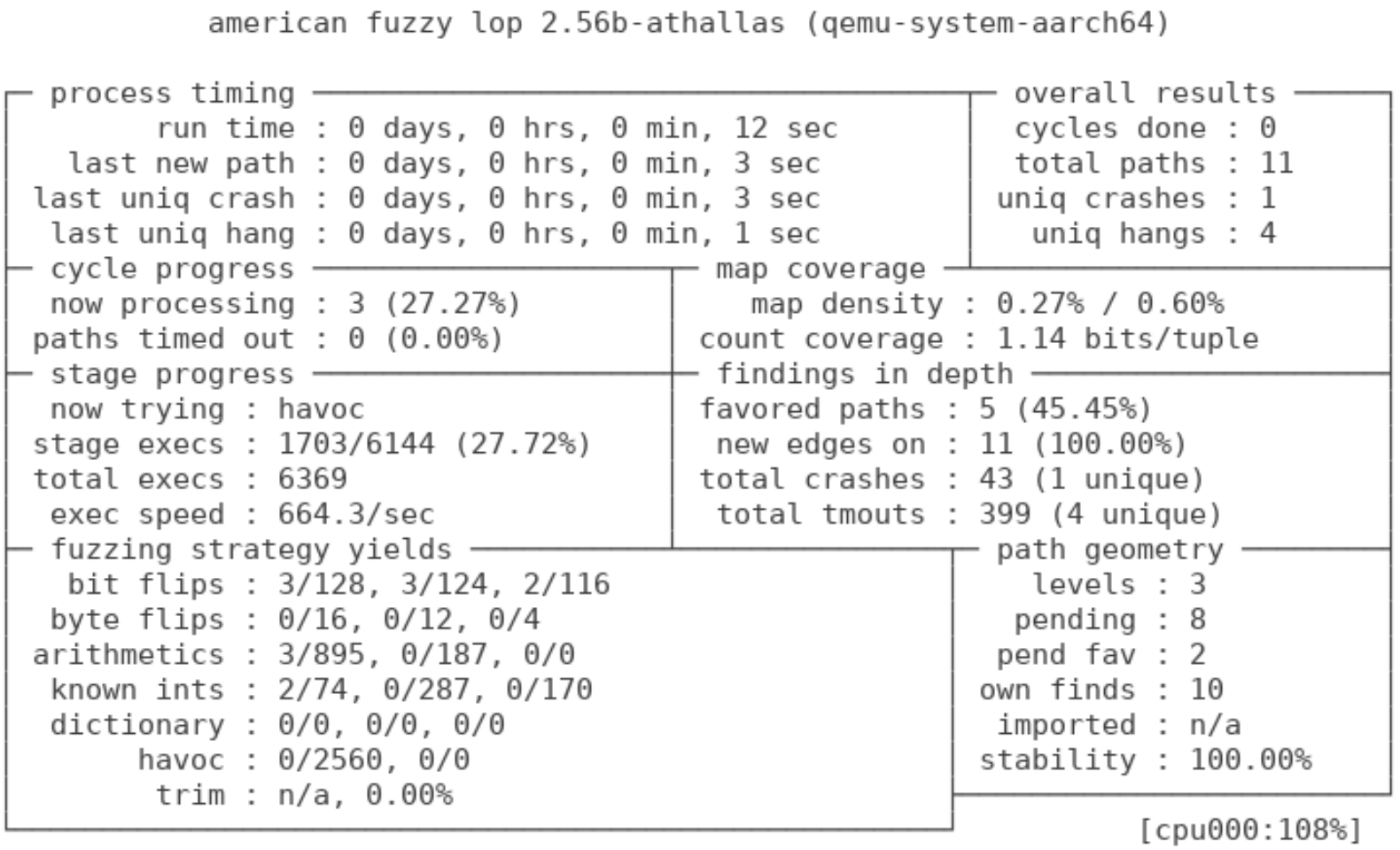
▷ References

- Lifting the (Hyper) Visor: Bypassing Samsung's Real-Time Kernel Protection, by *Gal Beniamini*, Project Zero
- Super Hexagon: A Journey from EL0 to S-EL3, By *Grant Hernandez*
- **Special thanks to huku**

▷ Teaser

- AFL with QEMU full system emulation
- Based on Triforce AFL
  - Project sadly abandoned
- Many QEMU changes since
- Versions used
  - QEMU v4.1.0
  - AFL v2.56b

▷ Teaser


american fuzzy lop 2.56b-athallas (qemu-system-aarch64)
process timing
run time : 0 days, 0 hrs, 0 min, 12 sec
last new path : 0 days, 0 hrs, 0 min, 3 sec
last uniq crash : 0 days, 0 hrs, 0 min, 3 sec
last uniq hang : 0 days, 0 hrs, 0 min, 1 sec
overall results
cycles done : 0
total paths : 11
uniq crashes : 1
uniq hangs : 4
cycle progress
now processing : 3 (27.27%)
paths timed out : 0 (0.00%)
map coverage
map density : 0.27% / 0.60%
count coverage : 1.14 bits/tuple
stage progress
now trying : havoc
stage execs : 1703/6144 (27.72%)
total execs : 6369
exec speed : 664.3/sec
findings in depth
favored paths : 5 (45.45%)
new edges on : 11 (100.00%)
total crashes : 43 (1 unique)
total tmouts : 399 (4 unique)
fuzzing strategy yields
bit flips : 3/128, 3/124, 2/116
byte flips : 0/16, 0/12, 0/4
arithmetics : 3/895, 0/187, 0/0
known ints : 2/74, 0/287, 0/170
dictionary : 0/0, 0/0, 0/0
havoc : 0/2560, 0/0
trim : n/a, 0.00%
path geometry
levels : 3
pending : 8
pend fav : 2
own finds : 10
imported : n/a
stability : 100.00%
[cpu000:108%]

Thank you!

CENSUS

IT Security Works