Nailgun Attack Lab

1 Introduction

NAILGUN [1] shows a severe vulnerability of the new debugging mechanism on the Arm architecture. Although the device manufacturers can defend against the attack by hardware-based modification (e.g., implementing a hardware-assisted control of the debug registers), they must launch a great callback of the affected devices, which triggers an unacceptable expense. To avoid the huge cost, we should design a software-level defense for NAILGUN attack. Then, deploying the defense can be implemented by a patch update via network, rather than the expensive callback.

To achieve the defense, we first decide where to place it. Considering that the attacker controls the kernel (i.e., the Operating System), we must leverage a higher privilege to monitor or prevent the NAILGUN attack. i.e., the Secure layer or the Hypervisor layer. In this lab, we select the Hypervisor layer to deploy the defense. Specifically, we introduce an address translation, called Stage-2 translation. By configuring the translation regime, we prevent the access of the registers from the kernel-level attacker, while the access of other memory regions is unaffected.

We consider the tasks of this lab as follows:

- Understand partial components of Armv8-A architecture, including the exception levels (EL) and the translation regimes.
- Understand how to use Raspberry PI 3 Module B+, and learn to burn the Linux kernel.
- Design a defense of NAILGUN, and implement it by modifying the Linux kernel.

In this lab, you are required to submit a report including **3** questions with 60% and 40% points, respectively. The question will be raised in the following sections.

2 Background

Before you start the lab, we strongly recommend you read this section.

2.1 Your Tools

2.1.1 Hardware

You have a Raspberry Pi. The Raspberry PI 3 Module B+ contains 4 Cortex-A53 cores, which is Armv8-A architecture. It supports both 32-bit Armv8-A (also called aarch32) and 64-bit Armv8-A (also called aarch64) architecture. The official kernel is compiled as 32-bit Armv8-A architecture.

2.1.2 Boot directory

In the SD card file system, it contains a directory, ”boot”, which stores the important configurations (”config.txt”), kernel (”kernel.img” or ”kernel7.img”), device tree files (”*.dtb”), and etc. In this lab, you should replace the **kernel** and necessary **device tree files**. You may use the configurations to support HDMI.

2.1.3 Source Code of Linux Kernel

In the following instructions, you should download the source codes of the Linux kernel, and compile them. You can download it here (version 4.14):

```
git clone http://github.com/raspberrypi/linux -b rpi-4.14.y
```

2.1.4 (Optional) Cross-compile Tools

You can compile the kernel on a virtual machine, then copy kernel image, dtb files, and modules into the disk. However, since you want to compile a `Arm`-based kernel, and your virtual machine is `x86` architecture, you must use the **Cross Compile** tools to achieve your goal.

Here is the source of the tools:

```
git clone git://github.com/raspberrypi/tools.git
```

When you `make` the kernel in 64-bit Ubuntu, you must configure the position of your compilation tools (`CROSS_COMPILE`).

2.2 Armv8-A Exception Levels

In Figure 1, Armv8-A define four exception levels (EL0 - EL3) with different privilege, and the higher number indicates the higher privilege. The components with higher-level privileges can access the source (e.g., memory and registers) of lower-level privileges. Detailed usage of exception levels are listed as follows:

- EL0: used for user applications, such as a game.
- EL1: used for the kernel, including the GPU driver, virtual address management, etc.
- EL2: used for the hypervisor, also called as virtualization layer.
- EL3: used for secure monitor (not used in our defense).

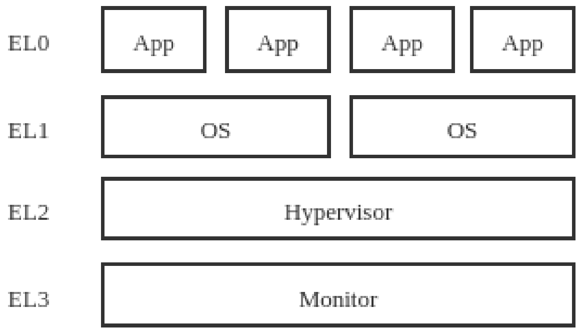


Figure 1: Armv8-A Exception Level, source from https://developer.arm.com/documentation/den0024/a/Fundamentals-of-ARMv8

Generally, if the lower-level user wants to use the feature in higher exception levels (for instance, an application requires a memory region to place the data). It will generate an exception (this is not a bad word), which will be handled by the corresponding exception handler in the higher ELs. Usually, such procedure is safe, because the normal system should not leave a ”backdoor” (such as providing a root privilege for the user) in the exception handlers. However, the NAILGUN leverages the debugging mechanism to jump to higher exception levels and execute arbitrary codes which do not exist in any exception handlers. Therefore, we regard the behavior as an ”attack”.

In this lab, we assume the attacker controls EL0 & EL1, and we design a defense on EL2. Note that the EL1 attacker cannot directly access the resource in EL2 (You can try to read an EL2 register in a kernel module, and find the module is crashed).

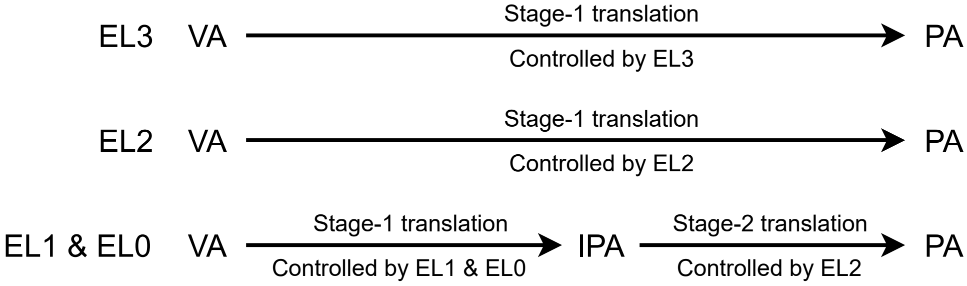


Figure 2: The mechanism of address translation in Armv8-A

2.3 Armv8-A Address Translation

For memory management, Armv8-A defines 3 types of address: the virtual address (`VA`), the intermediate physical address (`IPA`) and the physical address (`PA`). Armv8-A defines Stage-1 address translation for each exception level. Moreover, Armv8-A introduces an additional address translation for translation regime in EL0 & EL1, called the State-2 translation, which is controlled by EL2. As described in Figure 2, the VA in EL0 & EL1 must be first translated into an IPA before reaching a PA.

If the IPA to PA is failed, the translation will not reach a correct result. Therefore, in this lab, we can leverage the Stage-2 translation to control the access of the physical memory regions. Specifically, the region mapped to the debug registers.

3 Implementation

3.1 Compile the Kernel

This section tells you how to build a Linux kernel on Raspberry Pi. Specifically, we consider building (1) a Linux kernel with the defense mechanism, and (2) device tree files (dtb files) that contain the memory layout.

We strongly recommend you to prepare a **copy** of ”Raw” kernel and dtb files on disk or other devices. Therefore, when you build a corrupted kernel in the following steps, you can restore it quickly.

Note that we will give the instructions of ”compiling the kernel on Virtual Machine” instead of ”compiling the kernel on Raspberry Pi”. Actually the latter choice does not require the Cross-compile tools, but it is TOO SLOW.

3.1.1 Compile

Once you download the kernel, you can compile it. You enter the linux file and `make`. Since Raspberry PI Module 3B+ mainly supports the 32-bit Armv8-A architecture, we advise you define the architecture (`ARCH`) as `arm` (not `arm64`).

To compile it on Raspberry PI, you firstly prepare the configurations. Here we select a default bcm2709_defconfig. In parameter `CROSS_COMPILE`, you should provide the position of the downloaded cross-compile tools.

```
make -j8 ARCH=arm
   CROSS_COMPILE=tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian-x64\
/bin/arm-linux-gnueabihf- bcm2709_defconfig

make -j8 ARCH=arm
   CROSS_COMPILE=tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian-x64\
/bin/arm-linux-gnueabihf- menuconfig
```

Then, we will build the following things: (1) Image file (zImage) (2) device tree files (dtbs) (3) modules In parameter `INSTALL_MOD_PATH`, you should provide the position of the compiled modules.

```
make -j8 ARCH=arm
   CROSS_COMPILE=tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian-x64\
/bin/arm-linux-gnueabihf- zImage dtbs modules

make -j8 ARCH=arm
   CROSS_COMPILE=tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian-x64\
/bin/arm-linux-gnueabihf- modules_install INSTALL_MOD_PATH=modulespath
```

Be careful, the compilation may be stopped due to the lack of some essential tools (e.g., flex and bison), please download them with `apt-get`.

3.1.2 Replace

After you build the kernel, you can find the image in linux directory, which is

```
arch/arm/boot/zImage
```

And the device tree files in

```
arch/arm/boot/dts/
```

To replace the kernel and device tree files in the disk, we

- connect the Raspberry SD card to your computer.
- make a linux kernel by script tool
- copy your kernel, device tree files and modules to the `boot` directory of the SD card

The following commands can be helpful, note that (1) You should provide the position of the `boot` directory in your computer. (2) modules are unnecessary to copy to the `boot` directly, you can copy it to the `rootfs`

```
./linux/scripts/mkknlimg ./linux/arch/arm/boot/zImage boot/kernel7.img
cp ./boot/kernel7.img ./boot/kernel.img
cp ./linux/arch/arm/boot/dts/bcm2710-rpi-3-b-plus.dtb boot/
cp ./linux/arch/arm/boot/dts/overlays/*.dtb* boot/overlays/
mkdir rootfs/home/pi/modulespath
cp ./modulespath rootfs/home/pi/modulespath
```

Once you replace the kernel and reboot it successfully, you can use the command ”`uname -r`” to check the version of the kernel.

3.1.3 About the Nailgun module

Once we replace the kernel, we suggest building the Nailgun module with the compiled modules, or it would trigger a conflict with the version of the kernel.

You can compile it on your VM with modulespaths and cross-compile tools, then copy the .ko file to Raspberry Pi SD card.

```
obj-m += nailgun.o
obj-m += directly_read.o


KERNELDIR := modulespath/lib/modules/4.14.114-v7+/build



all:
make ARCH=arm -C $(KERNELDIR) M=$(PWD)
    CROSS_COMPILE=tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian-x64\
/bin/arm-linux-gnueabihf- modules

clean:
make -C $(KERNELDIR) M=$(PWD) clean
```

Question 1: Can you prove that (1) you have replaced the kernel (with ”uname -r” or other approaches), and (2) you have built the nailgun module with new headers? Please provide a figure. (**30%**)

Question 2: Can you run the Nailgun Attack on your new kernel? Please provide a figure. You can use ”dmesg” to show the execution result of Nailgun Attack. (**30%**)

3.2 Implementation of the Defense

After you understand the steps of building a kernel, we start to implement the defense. As Mentioned in Section 1, we leverage the Stage-2 translation to prevent the access of the memory-mapped interface for the debug registers. In Raspberry PI 3 Module B+, one address space of the registers is `0x40030000 - 0x40030fff`. Therefore, in Stage-2 translation, we protect the access of this region from the kernel and user applications.

Detailed Steps are provided as follows:

3.2.1 Memory Allocation

We reserve a 2MB memory region for the usage of Stage-2 translation. For instance, we select `0x32000000 - 0x321fffff`. Note that the region is large enough to store the page tables.

To reserve the memory, we should modify the corresponding device tree files.

We insert the codes in the device tree files (arch/arm/boot/dts/bcm2710-rpi-3-b-plus.dts)

```
   ...
   aliases {
      serial0 = &uart1;
      serial1 = &uart0;
   };

//----------insert codes start------------
   reserved-memory {
      #address-cells = <1>;
      #size-cells = <1>;
      ranges;

      test_reserved: test@32000000{
         compatible = "test,test-memory";
         reg = <0x32000000 0x200000>;
         no-map;
      };
   };
//-----------insert codes end--------------
};

&gpio {
   ...
```

Here we name the memory as `test@32000000`, while the start and size are provided in `reg`. The attribute `no-map` tells the kernel not to use the region when booting, so it protects the robustness of the system. However, if we do not implement any protection on this region, the kernel-level attacker can still access this region via some approaches.

3.3 Codes of Defense

3.3.1 Architecture of the codes

We begin to modify the source files of the Linux kernel. The source file of booting CPUs is `arch/arm/kernel/head.S`. You can find the `ENTRY(stext)`, which contains the configurations and codes for booting the primary CPU, and `ENTRY(secondary_startup)` for booting the other CPUs.

These codes have not implemented the Stage-2 translation, so we should achieve two goals:

- Creating a translation table, which entries are stored in the reserved memory (`0x32000000 - 0x321fffff`)
- Configure the system registers to enable the Stage-2 translation

Since we are programming with the assembly language, we should use some regular registers to execute the add/load/store instructions. However, the kernel codes may occupy several registers to store some information (e.g., the branch address in the following steps). If we want to use them, we dump the values in the registers into memory and put them back later. It can be achieved by a temporarily used register.

Our code is placed between the comments `Our codes start` and `Our codes end`. We place the codes in two postions (primary and secondary CPUs), and you can also place your codes in other positions. Note that the codes for primary CPU is not necessary in the Nailgun example, since it use core 1 (a secondary core) to map the debugging registers. In our example, we can only implement the Stage-2 translation on the secondary cores.

```
ENTRY(stext)
ARM_BE8(setend be )        @ ensure we are in BE8 mode

THUMB(   badr  r9, 1f       ) @ Kernel is always entered in ARM.
THUMB(   bx r9    )  @ If this is a Thumb-2 kernel,
THUMB(   .thumb         )  @ switch to Thumb now.
THUMB(1:       )


#ifdef CONFIG_ARM_VIRT_EXT
bl __hyp_stub_install
#endif

/*Our codes start*/
...
/*Our codes end*/

   @ ensure svc mode and all interrupts masked
safe_svcmode_maskall r9

...

ENTRY(secondary_startup)
/*
* Common entry point for secondary CPUs.
*
* Ensure that we're in SVC mode, and IRQs are disabled.  Lookup
* the processor type - there is no need to check the machine type
* as it has already been validated by the primary processor.
*/

ARM_BE8(setend be)            @ ensure we are in BE8 mode

#ifdef CONFIG_ARM_VIRT_EXT
bl __hyp_stub_install_secondary
#endif

/*Our codes start*/
...
/*Our codes end*/

   safe_svcmode_maskall r9

mrc   p15, 0, r9, c0, c0      @ get processor id
bl __lookup_processor_type
movs  r10, r5
....
```

Here is an example, you can also use other general registers which is temporarily used. You can find that we store the values of general registers in our reserved memory, since they are used in the next codes. To distinguish whether the register will be used or not, please read the source code of the kernel.

```
//----------insert codes start------------
   /*Here we can use the regular registers r0
   *and we store the value r1, r2, r3, r4, r5
   *in our reserved memory
   */
   ldr r0,=0x32000100
   str r2,[r0]
   ldr r0,=0x32000104
   str r3,[r0]
   ldr r0,=0x32000108
   str r4,[r0]
   ldr r0,=0x3200010C
   str r5,[r0]
   ldr r0,=0x32000110
   str r1,[r0]
   /*creating page table*/
   ...
   /*configuring system registers*/
   ...

   /*We finally fetch the values*/
   ldr r0,=0x32000100
   ldr r2,[r0]
   ldr r0,=0x32000104
   ldr r3,[r0]
   ldr r0,=0x32000108
   ldr r4,[r0]
   ldr r0,=0x3200010C
   ldr r5,[r0]
   ldr r0,=0x32000110
   ldr r1,[r0]
   mov r0,#0 //restore r0
//-----------insert codes end--------------
```

Now we can use the registers r0,r2,r3,r4,r5 to achieve the defense mechanism.

3.3.2 Creating Translation Table

Design of the Stage-2 translation table without defense

In this step, we fill the translation table into the reserved region. Simply, we create a flat mapping between the IPA and PA of the whole address space (i.e., IPA == PA), and invalidate the mapping of both the debug registers (0x40030000 - 0x40030fff) and the Stage-2 translation table (0x32000000 - 0x321fffff).

A completed process of Stage-2 translation is separated into several levels (See in Figure 3). For each level, the MMU will combine the input address with the entry (or the base address) at the current level, and get the address of the entry at the next level.

Structure of the table entry (descriptor)

In Figure 4 and Figure 5 (you can find the attributes in page G5-6290 and G5-6291), each entry (or called as a descriptor) indicates the attributes (e.g., read permission, write permission, and access permission) of

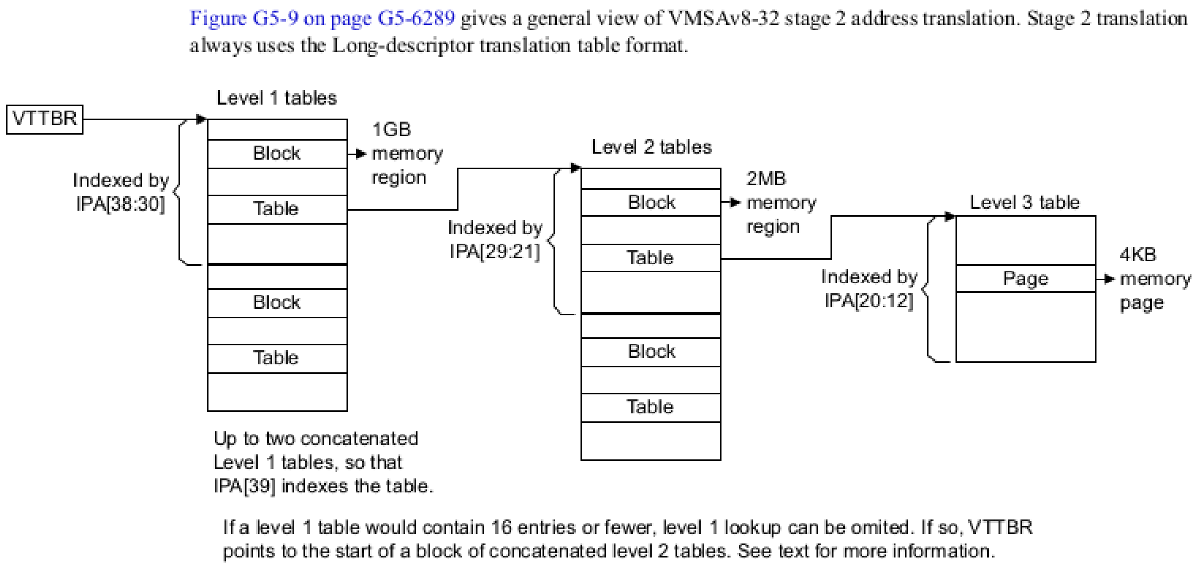


Figure G5-9 on page G5-6289 gives a general view of VMSAv8-32 stage 2 address translation. Stage 2 translation always uses the Long-descriptor translation table format.

If a level 1 table would contain 16 entries or fewer, level 1 lookup can be omited. If so, VTTBR points to the start of a block of concatenated level 2 tables. See text for more information.

Figure 3: Stage-2 translation overview

specific address space. You can also find the partial component of the address related to the next-level entry (descriptor) or the output. In particular, we care about the last two bits (bit[1:0]) of the entry, because they will tell us whether the translation should continue or stop.

- If we find a block or page, the translation is finished and will return the value.
- If we find an invalid entry, the translation is finished and will return a fault.
- If we find a table, the translation should continue.

Example of a table walk

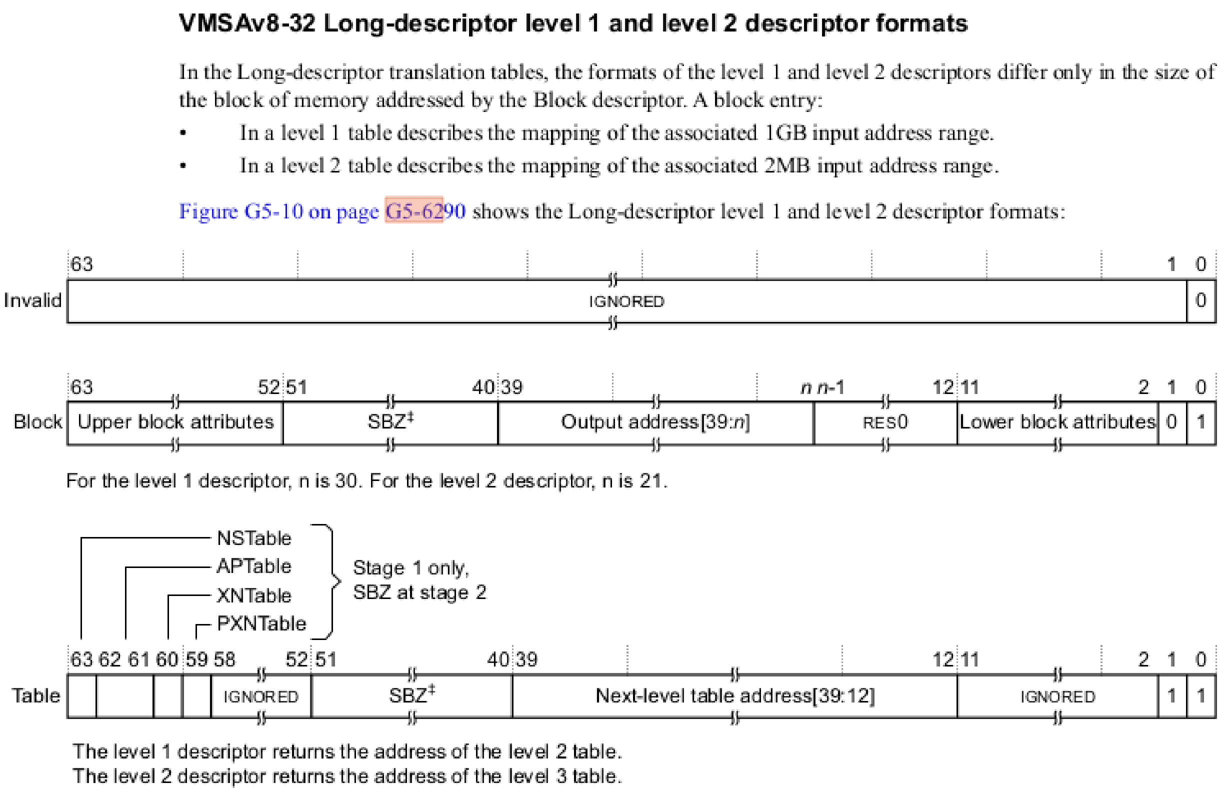
Here we provide an explanation of a translation starting at level 1 in Figure 6. The corresponding Picture is provided in K7-8498 on the Armv8-A manual since Stage-2 translation implements the Long-descriptor format on the Aarch32 translation regime. Note that n=5 in the picture.

When MMU receives the translation requirement and the input IPA, it separates the IPA into 4 parts for the use of 3 levels. In level 1, MMU combines the first part of input IPA and the base address of the Stage-2 translation table, then calculates the address of the level-1 entry. The last two bits of the entry tell MMU whether the translation should continue, stop or return a fault. If we continue the translation, MMU combines the second part of the IPA with the level-1 entry to get the region of the address of the level-2 entry. Finally, the level-3 entry tells the translation is valid or not. If the translation is valid, MMU combines the level-3 entry and the last component of the IPA (we can regard it as offset) to get the output result.

Example of the codes

Here we assume the base address of the Stage-2 translation table as 0x32000000 and give an example of the Stage-2 translation for the region 0x80000000 - 0xbfffffff.

```
ldr r0,=0x32000010
```



VMSAv8-32 Long-descriptor level 1 and level 2 descriptor formats

In the Long-descriptor translation tables, the formats of the level 1 and level 2 descriptors differ only in the size of the block of memory addressed by the Block descriptor. A block entry:

- In a level 1 table describes the mapping of the associated 1GB input address range.
- In a level 2 table describes the mapping of the associated 2MB input address range.

Figure G5-10 on page G5-6290 shows the Long-descriptor level 1 and level 2 descriptor formats:

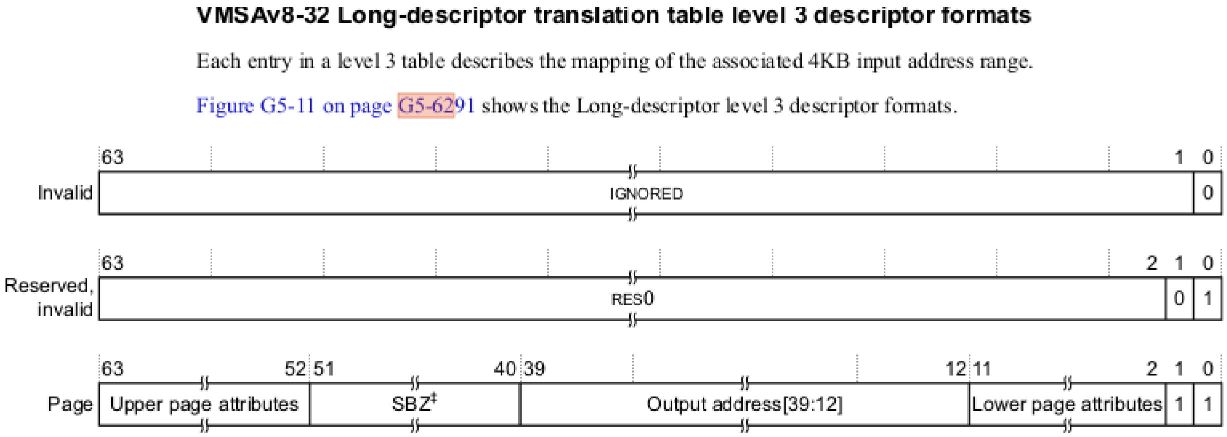
Figure 4: Structure of the level 1 & 2 table entry (descriptor)

```
ldr r1,=0x800007FD
str r1,[r0]
add r0,  r0,   #4
ldr r1,=0x00400000
str r1,[r0]
```

The above codes mean that we store the entry 0x0040_0000_8000_04FD in the region 0x32000010 - 0x32000017. The last 2 bits indicate it as a block, which means the translation is finished here. For a input IPA 0x81234567, the level 0 will combine bit[31:30] of IPA and bit[31:5] of the base address, then tell we to read the address 0x32000000 + (0x2 <<3) = 0x32000010. Then, we get the level-1 entry by reading the value in this address. With the bit[1:0] of this entry, we know it is a block (you can see Armv8-A manual on page G5-6290), not a table. So, our translation is finished, the output address is composed of two parts. Its bit[31:30] is from the entry, and bits[29:0] is from the input address. You can calculate it, and know that the value of the output address is the same as the input address.

Not done yet

Although the given example is easy to follow, we must configure the corresponding entries with smaller granules (i.e., the translation should not stop in level 1), since we consider protecting a 4KB region and a 2MB region, rather than a 1GB region. To protect the 2MB region (it is the Stage-2 translation table, 0x32000000 - 0x321fffff), we should separate the 1GB region, 0x0 - 0x3fffffff into 512 2MB-sized regions. If we want to protect the 4KB region (it is the debug registers, 0x40030000 - 0x40030fff), we should separate the 2MB region, 0x40000000 - 0x401fffff into 512 4KB-sized regions. If not, we may protect an unexpected



VMSAv8-32 Long-descriptor translation table level 3 descriptor formats

Each entry in a level 3 table describes the mapping of the associated 4KB input address range.

Figure G5-11 on page G5-6291 shows the Long-descriptor level 3 descriptor formats.

Figure 5: Structure of the level 3 table entry (descriptor)

address space.

Defense implementation

Finally, we implement the protection of these regions by setting the last bit of the corresponding entry as 0, because in Figure 4 and 5, if the last bit (bit[0]) of an entry is 0, then it indicates an invalid entry. Therefore, when the MMU performs the Stage-2 translation for the access of these regions, it finally finds an invalid entry and returns a translation fault.

3.3.3 Configuring the System Registers

We consider to configure three significant registers: `VTTBR`, which indicates the base address of the Stage-2 translation; `HCR`, which enables the Stage-2 translation; `VTCR`, which indicates some attributes of the translation. If you are interested in the functions of such system registers, please read the reference manual.

First, we fill the `VTTBR` register, we directly put the start address (0x32000000) into the `BADDR` bits:

```
ldr r0,=0x32000000
ldr r1,=0x0
mcrr p15, 6, r0, r1, c2
```

We then fill the `VTCR` register. We mainly care about the `T0SZ` and `SL0` bits, which indicate the region size and starting level, respectively:

```
ldr r1,=0x80000040
mcr p15, 4, r1, c2, c1, 2
```

Finally, we configure the last bit (`VM`) of the `HCR` as 1 to enable the Stage-2 translation:

```
mrc p15, 4, r0, C1, C1, 0
orr   r0, r0, #0x1
mcr p15, 4, r0, C1, C1, 0
```

Question 3: Can you explain the process of Stage-2 translation with your translation table? Here we require you to translate an IPA, `0x40030123`, to the identical value of PA. In this question, you should provide (1) your files of Stage-2 translation, which should be different from the provided files, and (2) the

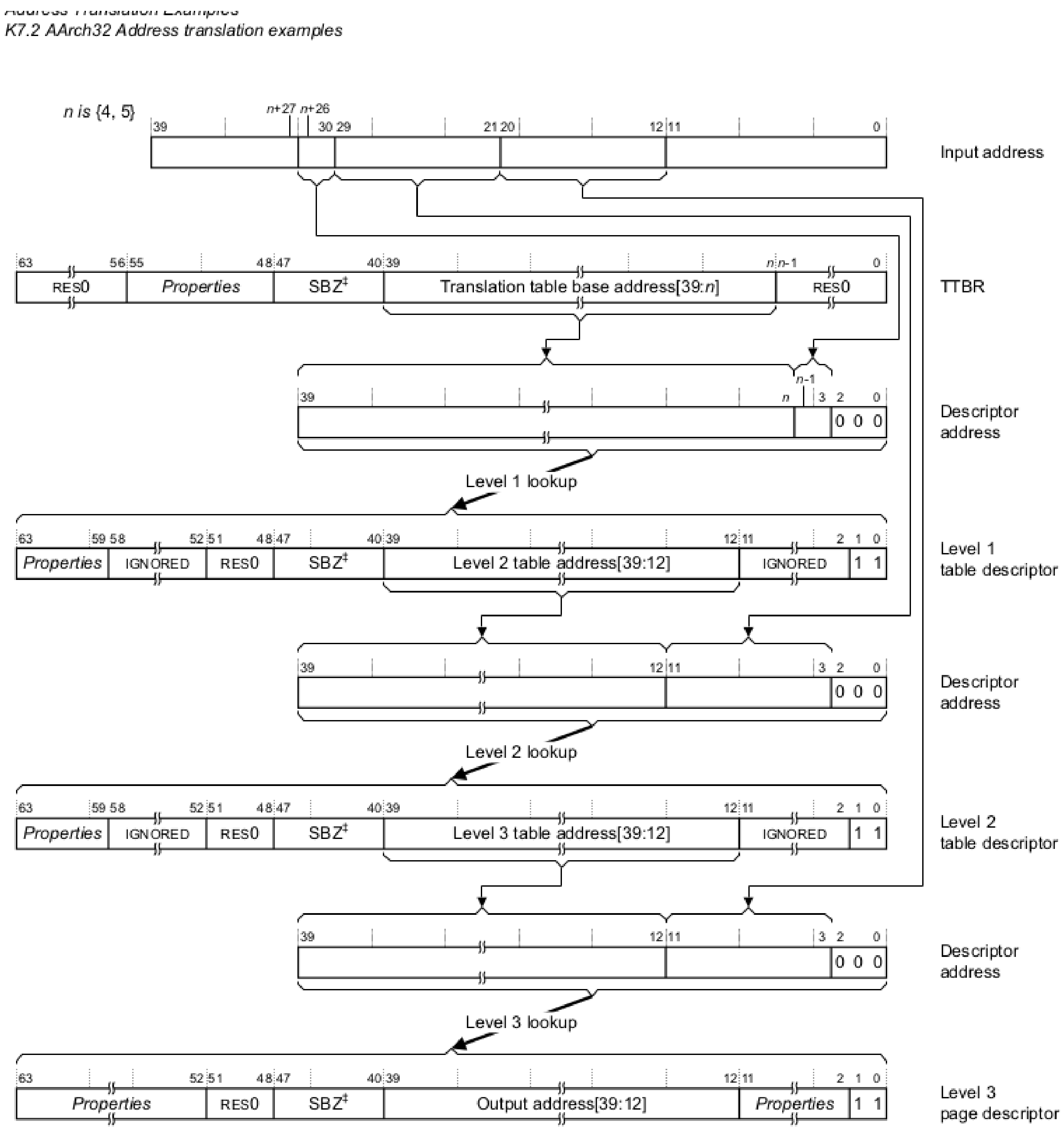


Figure 6: A translation example

explanation for table walk, which should tell us the `address` and `value` of the descriptors (page table entry). (**40%**)

3.4 Other Useful Techniques

3.4.1 Change Download Source

To download tools with `apt-get`, we suggest you change the source. Here is one possible source.

First, open the `/etc/apt/sources.list`, change the content as

```
deb https://mirrors.tuna.tsinghua.edu.cn/debian/
buster main contrib non-free

deb-src https://mirrors.tuna.tsinghua.edu.cn/debian/
buster main contrib non-free
```

Then, open the `/etc/apt/sources.list.d/raspi.list`, change the content as

```
deb http://mirrors.tuna.tsinghua.edu.cn/raspberrypi/
buster main ui
```

Finally, update the keys and the content. Note that you should select one available `recv-keys` on the key server.

```
apt-key adv --keyserver keys.gnupg.net --recv-keys xxxxxxx
apt-get upgrade
apt-get update
```

4 Evaluation

4.1 Effectiveness Evaluation

To prove the effectiveness, we first load a Nailgun kernel module on the Raspberry PI to prove the vulnerability. Then, we burn the kernel with the defense on the same Raspberry PI and load the Nailgun kernel module.

Once the module is loaded, the system will **hang** since (1) it generates a translation fault or so-called as an **exception** (2) the corresponding exception handler asks to jump to the current address (in assembly language is ”`b .`”), and triggers an endless loop. The Linux kernel does not process the exceptions well in EL2 or EL3.

4.2 Performance Evaluation

As we said in Section 1, we only leverage the Stage-2 translation that generate **negligible** performance overhead. To prove it, you can run several **benchmarks** on the native system and the defense-enabled system. For instance, the *Sysbench* [2]. You can find the website with the manual in the Reference. You can also run other benchmarks to measure the performance.

References

[1] Z. Ning and F. Zhang, “Understanding the security of arm debugging features,” in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 602–619.

[2] akopytov, “sysbench,” https://github.com/akopytov/sysbench.