Trusted Execution in ARM

Raz Ben Yehuda

Prof. Nezer Zaidenberg.
Dr. Amit Resh .
Dr. Michael Kipperberg.

Abstract

TrulyProtect [1], Microsoft device guard [2], SGX[3] and others use the Hypervisor as a security technology. ARM's Hypervisor is a virtualization extension technology [4]. As ARM architecture is widely deployed in the mobile, homeland security, IoT, autonomous cars and other industries We present a method to apply TrulyProtect thin Hypervisor technology as a generic security solution independent of the operating system and architecture. In this paper we will introduce TrulyProtect thin Hypervisor to the ARM platform and provide a performance benchmark.

Introduction

TrulyProtect thin Hypervisor is an implementation of the blue pill concept [5] for ARM. It is a thin layer of code which is invoked through traps (Exceptions in ARM) from a lower priority levels.

The thin Hypervisor is implemented in an ARMv8-a 64bit processor. The Hypervisor can be considered as Trusted Execution Environment (TEE) in the sense that it offers an isolated execution environment of a decrypted code.

This paper describes how an Truly’s Hypervisor is used to provide TEE. What are the downsides, what can be improved and what is expected next.

ARMv8-a Hypervisor

ARM architecture has four privileges levels, EL0,EL1,EL2 and EL3. EL0 privilege level is used to execute user space programs, EL1 is mostly used to execute kernel space, EL2 is used to execute Hypervisor code. EL3 , if used, contains the TrustZone firmware.

The Hypervisor is a piece code which is loaded as a vector of execution entries. The vector address is kept in a register called VBAR_EL2. This vector holds 16 execution starting addresses, each of which is invoked when certain events take place..

Hypervisor installation

When a processor boots the register VBAR_EL2 is not initialized. In order to fill it a code must execute in EL2 or EL3. For this reason, the Linux kernel is booted into EL2 mode and initializes VBAR_EL2 with an initial vector. This vector main use is to provide the facility to be replaced with a new vector. In the Linux case, KVM installs its vector. Truly puts its own vector instead of KVM’s.

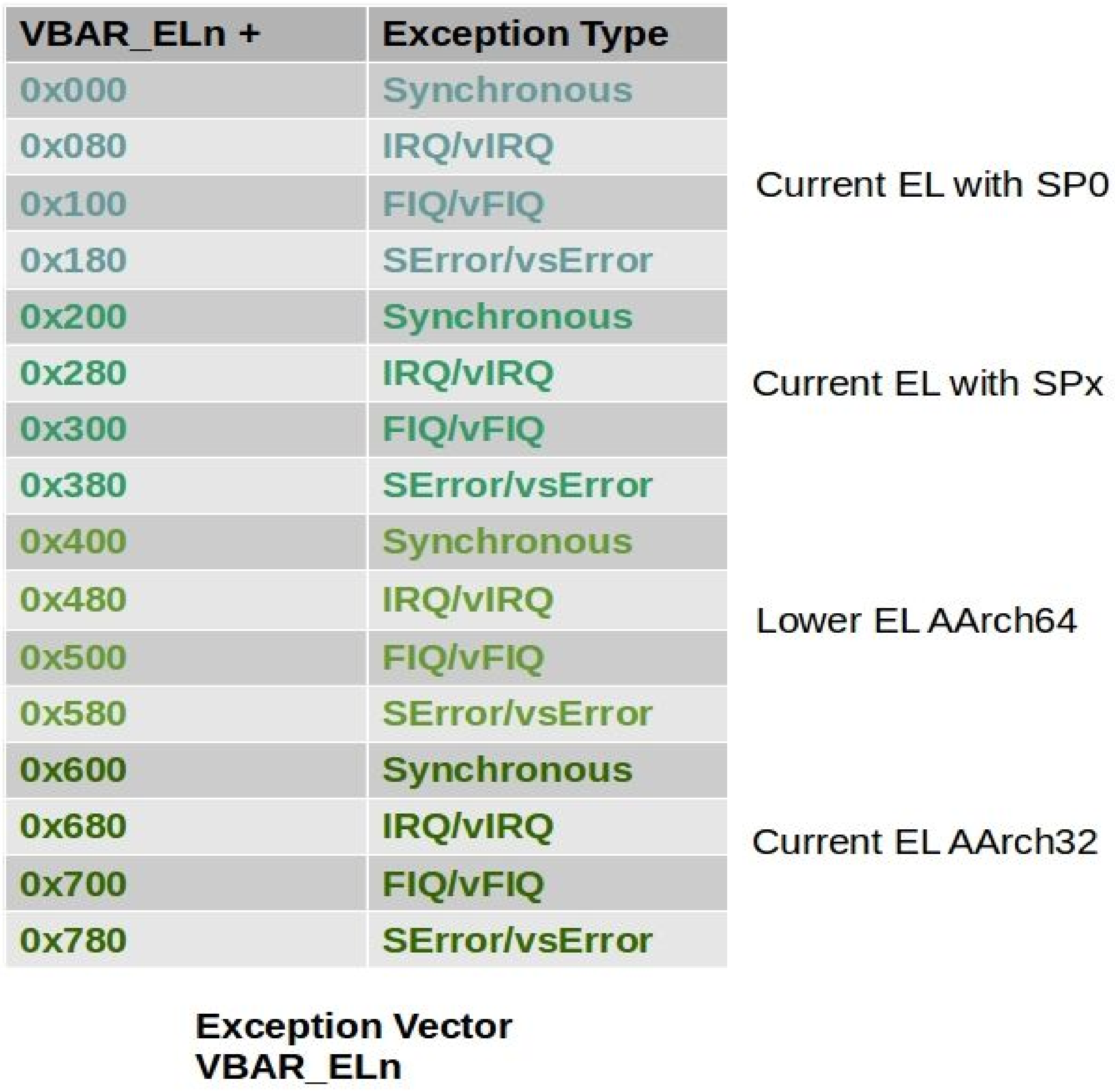
.

Hypervisor vector format

ARMv8 architecture presents three exception vectors for each of the three privileged levels, for EL1,EL2 and EL3. They are identical in their formation. Each vector is composed from 16 entries, consolidated in 4 groups. The entries are not pointers to functions but the actual blocks of code so they are restricted in their size.
The vector formation is depicted in the below Figure (Figure 1). Truly implements only two entries. These entries are explained in detail later in this paper.

The first four entries, or the first group are exceptions which are invoked whenever an exception takes place and the executing code was using the SP0 stack (user space stack) while it was executing in EL2 (current EL). In this group Truly implements the first entry.

The second entry that Truly implements is the first function of the third group (at offset 0x400). This function is invoked whenever a user calls **HVC** (a hypervisor call) from a lower level, mainly EL1, or when a predefined trap is being called from any other lower privilege level.
To return from an exception the **ERET** command is used.



VBAR_ELn +	Exception Type	
0x000	Synchronous	Current EL with SP0
0x080	IRQ/vIRQ	
0x100	FIQ/vFIQ	
0x180	SError/vsError	
0x200	Synchronous	Current EL with SPx
0x280	IRQ/vIRQ	
0x300	FIQ/vFIQ	
0x380	SError/vsError	
0x400	Synchronous	Lower EL AArch64
0x480	IRQ/vIRQ	
0x500	FIQ/vFIQ	
0x580	SError/vsError	
0x600	Synchronous	Current EL AArch32
0x680	IRQ/vIRQ	
0x700	FIQ/vFIQ	
0x780	SError/vsError	

Exception Vector
VBAR_ELn

Figure 1: EL2 Exception vector

ARM's Hypervisor memory explained

ARMv8 Hypervisor is designed to run in a separate address space than the other levels. This means that EL2 code may use a different translation table [9] .This table is referred to by a register called TTBR0_EL2. ARMv8 memory translation architecture defines two address space regimes – a single virtual address range and a two virtual address space ranges.

1. The first regime is of 48 bits and it ranges from 0x0000000000000000 to 0x0000FFFFFFFFFFFF.

2. The second regime is a virtual address that ranges from 0x0000000000000000 to 0x0000FFFFFFFFFFFF and another from 0xFFFF000000000000 to 0xFFFFFFFFFFFFFFFF.

EL1 translation regime is of the second type and it uses TTBR0_EL1 and TTBR1_EL1 as translation tables.
EL2 translation regime is of the first type. It uses TTBR0_EL2 register. This means that the address space of the Hypervisor cannot refer both to a Linux kernel address and a user space address at the same time so some modifications are made to support that (ARMv8a-1 added support for that , known as VHE – virtual host extensions [7]). This issue is discussed later in the paper.
ARM defines an IPA (Indirect physical address) as the OA (output address) of stage 1 translation and the IA (input address) of stage 2 translation.
A code of an Hypervisor exception vector must be pre-configured and be aligned to 2048 bytes. Also, the code must be mapped to the Hypervisor translation tables. To map any Hypervisor code in Linux we assign each function to a segment named hyp_text which is mapped to the Hypervisor translation tables during initialization. This

way, any C function prefixed by a “hyp_text” segment is mapped to the hypervisor and can be referenced and executed from the Hypervisor.

But in-order to invoke a C function or access a global variable from the hypervisor the address must recalculated. As noted earlier, a kernel address cannot be accessed directly because of the single address space regime and thus these addresses are mapped to a certain offset = 0x4000000000.

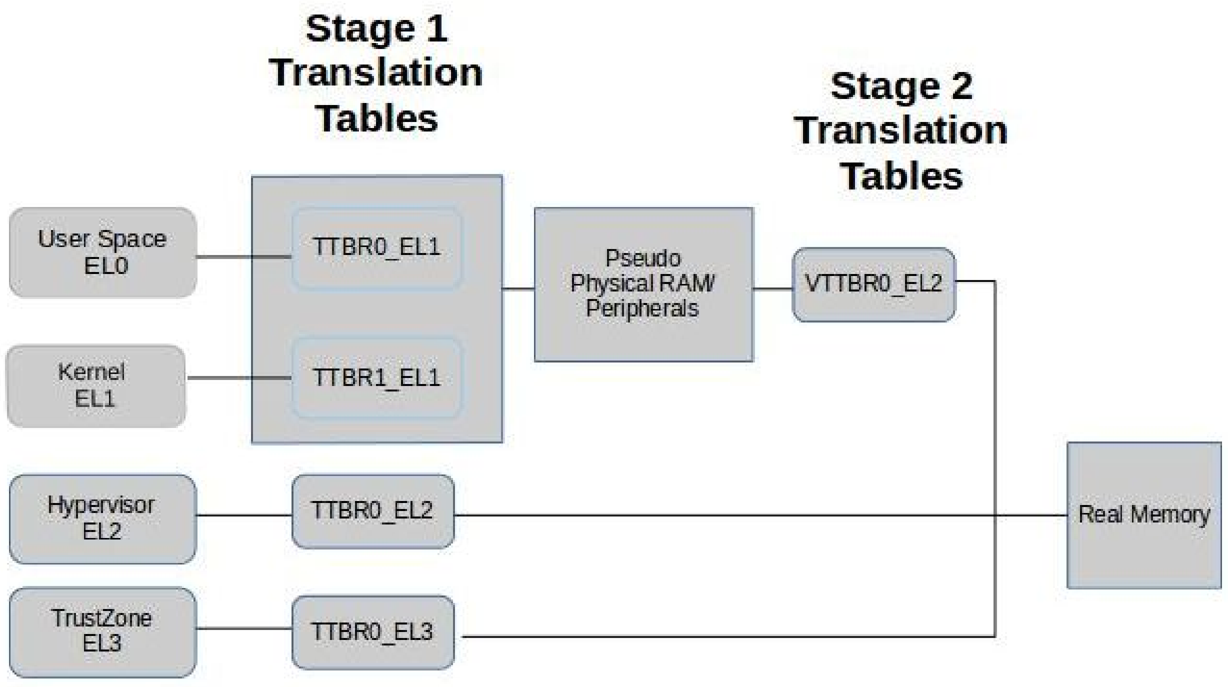


Figure 2: Translation regimes

Related technologies

Trust Zone

ARMv8 presented [11] TrustZone as part of its architecture. TrustZone can be used in two distinct operational modes:

1. A Monitor mode - A separate operating system that runs concurrently with a generic operating system. The processor assigns time to the TrustZone operating system from time to time.

2. A Passive library - A monitor exception vector which is being activated through traps or SMC calls.

It is possible to implement TrustZone as a virtual machine; However, TrustZone is designed to execute in an isolated environment in a higher privilege level than the hosting machine. This way a bug in the hosting machine cannot be exploited while in the VM it is possible.

Cells

Cells [12] is a virtualization technology for mobile phones. It does not use a hypervisor but provides isolation of a namespace. It separates parts of the file system and it excels in low memory consumption. Cells runs on ARMv7.

Truly Protect memory architecture

IPA

TrulyProtect sets a thin hypervisor that creates an IPA mapping during the kernel boot. The kernel is unaware that it moved from a single translation regime to a two stages translation regime.
TrulyProtect defines the IPA as follows: **IPA=PA.** This equality applies only to addresses. Truly does not change the mapping but the access rights. For instance, a translation table in stage 1 may define a page to be readable and writable but in stage 2 it may be only readable. This way a malicious user is prevented from accessing protected pages.

In ARM, a page descriptor in a stage 2 table differs from stage 1 translation. In general, page attributes are divided to lower and upper parts. Lower attributes are the lower 12 bits, and the upper attributes are the top 16 bits. These differences makes it impossible to just copy page descriptors from stage1 to stage2.

The starting address of the translation tables is cached in VTTBR0_EL2. The memory that holds the tables is a page that does not need to be mapped to the hypervisor. The register VTTBR0_EL2 caches the physical address of the starting page.

Memory Access

Truly uses the Hypervisor’s translation table (referenced by TTBR0_EL2) to map a user space memory and a kernel space memory.
At boot time, the initial exception vector is not activating the MMU. Later in the boot process the MMU is turned on.

At boot Truly maps only the virtual machine state structure. This structure is cached in the TPIDR_EL2 register of each processor.
Each time an encrypted process is loaded, it is essential to map parts of its code to the Hypervisor. User space code stack and global addresses must be mapped as they are without any additional offsets calculations because we do not want to modify the users program code to access addresses. For example, if a user decrypted code accesses address 0x400100, this address must be mapped to the hypervisor translation table as it is, without an offset.

However, since the memory accessed is both a user space memory and a kernel space memory, there can be address space collisions.

User space address are in the range of 0x0000000000000000 to 0x0000FFFFFFFFFFFF while kernel space addresses are in the range of 0xFFFF000000000000 to 0xFFFFFFFFFFFFFFFF . TrulyProtect uses Linux KVM solution for accessing kernel space memory. Linux KVM sets all kernel memory addresses by adding 0x4000000000 to the addresses. So, for instance, a user space address 0x400050000 can also be a kernel address 0x50000 mapped to the Hypervisor.

The way we chose to solve this possible collision is by creating a so called “fake vmas”. A vma [8] is virtual contiguous address space that all of its pages share the same attributes and access rights. Each page in a fake vma has all of its permissions bits set to non-accessible. This way we pre-allocate the addresses used by the Hypervisor *.

* Fake VMAs are not implemented at the time of writing of this article.

Protecting a program

An encrypted is a program that was processed by TrulyProtect StaticAnalyzer[14]. The StaticAnalyzer replaces a designated function with the “brk #3 “command and adds the encrypted function code at the end of programs. The brk command is used by the debugger to generate an exception.

For example, let us consider function for encrypting:

```
int foo() {
        return 19;
}
```

We compile it with g++:

```
_Z3foov():
  40bda8:     52800260      mov    w0, #0x13                           // #19
  40bdac:     d65f03c0       ret
```

And after processing it with the StaticAnalizer, all of the assembler commands are replaced by the brk assembler command.

```
_Z3foov():
  40bda8:     d4200060      brk    #0x3
  40bdac:     d4200060      brk    #0x3
```

Figure 3: Padded Function

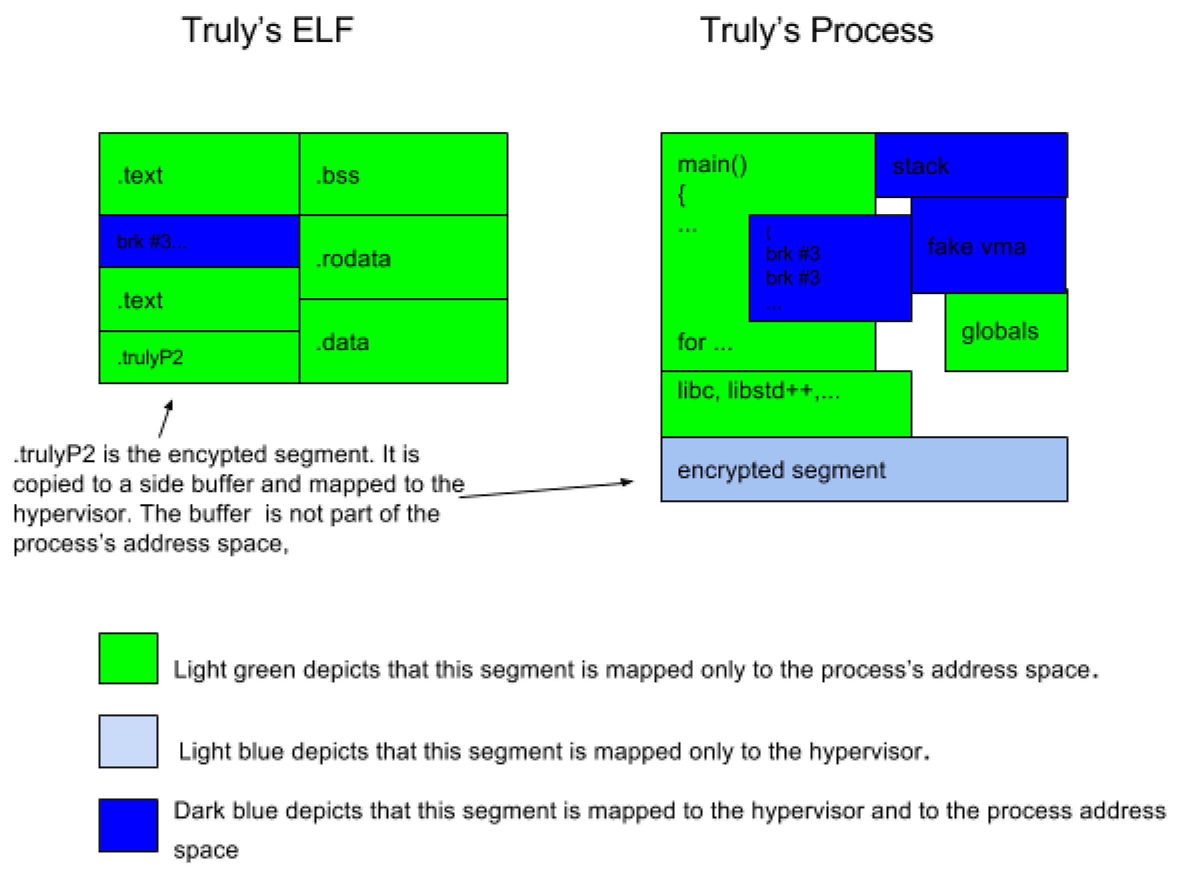
The brk command causes an exception when the ninth bit (0x100) of the mdcr_el2 register is set. And this bit is set only when an encrypted program is executed.

During the time of the execution it would not be possible to set a breakpoint in any other program, this is because when the Hypervisor is trapped it must jump over the brk

command or turn off the trap in the mdcr_el2 register. Thus, when trying to debug another program breakpoints will not work. This is one of the reasons we chose the brk command as it adds complexity of trying to debug the protected program.
In x86 we chose the hlt command. There is no exact hlt command in ARM, the closest commands are wfi (wait for interrupt) and and wfe (wait for exception),which moves the processor to a sleep state and until an exception or an interrupt happens. Though it is possible to use these commands they are extensively used by the Linux kernel. Each time a processor is put to sleep it uses these commands. So, if we trap wfi/wfe we change the expected behavior of the operating system. The Android kernel is a common example for which processors are turned off and on quite frequently.
Another reason for choosing the mdcr is that it is an independent mechanism to generate a trap and it is independent from the interrupt controller. This way we reduce the overhead of processing virtual interrupts.
Once the execution is completed then the ninth bit of mdcr_el2 register is turned off and one can debug programs regularly.

Mapping a process to the Hypervisor

As depicted in Figure 4, as soon as the encrypted program is detected, the pages that contain a copy of the encrypted function, the padded function and the fake vmas are mapped to the Hypervisor. The only memory allocation is the encrypted function so the memory footprint is the same size as padded function in pages granularity.



Truly's ELF
Truly's Process
.text
.bss
brk #3...
.rodata
.text
.data
.trulyP2
main()
{
...
stack
(
brk #3
brk #3
...
fake vma
globals
for ...
libc, libstd++,...
encrypted segment
.trulyP2 is the encypted segment. It is copied to a side buffer and mapped to the hypervisor. The buffer is not part of the process's address space,
Light green depicts that this segment is mapped only to the process's address space.
Light blue depicts that this segment is mapped only to the hypervisor.
Dark blue depicts that this segment is mapped to the hypervisor and to the process address space

Figure 4: Encrypted Process and Encrypted ELF

In figure 5,we see that once the program counter (instruction pointer) reaches the brk (1) instruction then an exception takes place (2). By examining the syndrome register (3) the Hypervisor checks whether the exception is caused by a brk and if a Truly program caused it. If not then it exits.
The verification of whether the program is Truly's is done by checking if the ttbr0_el1 holds the reference to the memory table of the protected process. In step 4 the Hypervisor decrypts the function on top of the padded code. In step 5 we make some preparations to jump to the decrypted code, mainly by setting the stack to the user space stack (SP0).

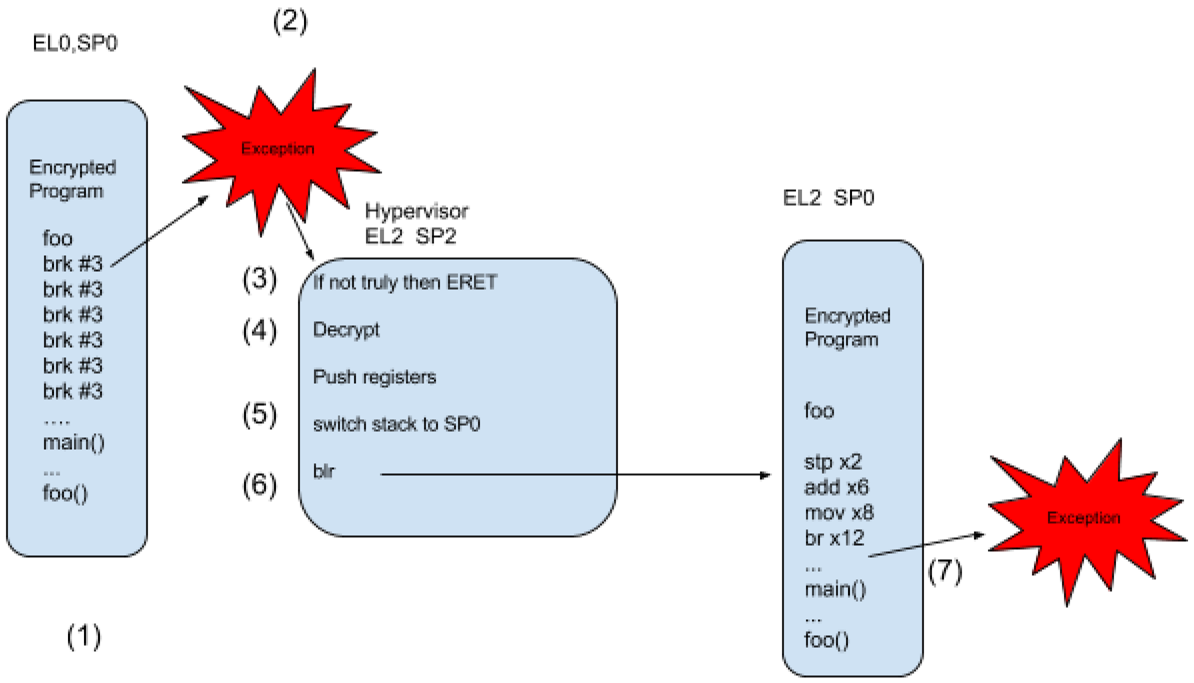


Figure 5: Trap from EL0 to EL2

In step 6 the Hypervisor actually starts executing the decrypted function code. It runs until the function ends or in the event of EL2 exception(7).

In Figure 6 an exception took place in EL2.

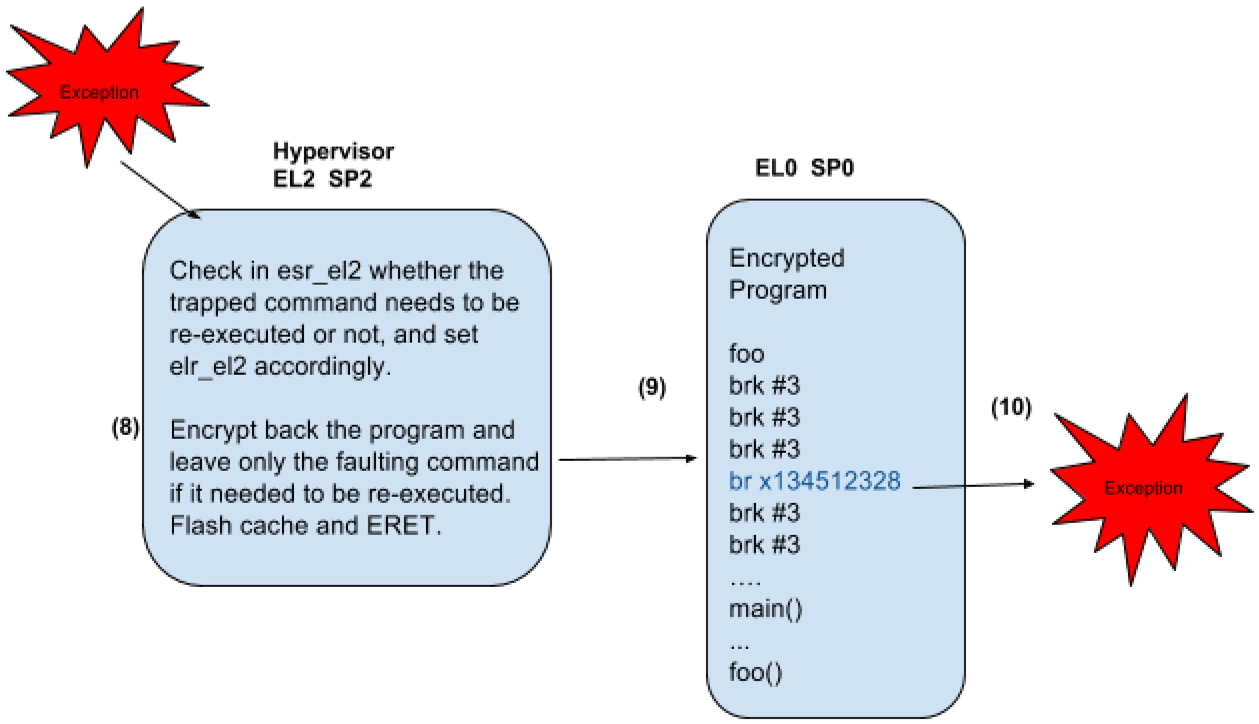

Exception
Hypervisor
EL2 SP2
Check in esr_el2 whether the trapped command needs to be re-executed or not, and set elr_el2 accordingly.
(8)
Encrypt back the program and leave only the faulting command if it needed to be re-executed. Flash cache and ERET.
(9)
EL0 SP0
Encrypted Program
foo
brk #3
brk #3
brk #3
br x134512328
brk #3
brk #3
....
main()
...
foo()
(10)
Exception

Figure 6: Trap from EL2 to EL2

In step (8) an exception had taken place and the execution cannot continue in EL2. The Hypervisor puts back the padded code except for the command that caused the exception. Then it sets elr_el2 to this address and ERETs. The elr_el2 register holds the address to return to after an ERET;

At this point (9) the program continues to run until it executes the brk command once again(10). A common case is that the program runs some code in libc and then returns back to padded function. Once this happens the entire cycle starts again until the function finishes if any.

Interrupts in EL2

A Processor executing code in EL2 privilege level may be interrupted by peripheral devices or other processors. This means that the thin Hypervisor needs to handle FIQs or IRQs exceptions. But handling interrupts in EL2 is a complicated task.So it would be best to disable interrupts. Disabling interrupts altogether also means disabling

exceptions and we cannot do that because the decrypted function code running in EL2 must be able to fault into EL2.
The best solution would be to disable interrupts only in EL2 while executing in EL2. For this reason (and others which beyond the scope of this paper) the HCR_EL2.TGE (Trap General Exceptions) is zero. When HCR_EL2.TGE is zero it means (Table D1-13 in [4]) that when a code executes in EL2 and the exception cannot be taken then the trap is put to pending mode. Once the processor changes to EL1 mode then the interrupt would be served.

Disabling interrupts for long durations is considered a bad practice. As noted before the thin Hypervisor maps only the encrypted code memory so that the code would run as less as possible in EL2. This way we reduce the amount of time we execute with interrupts disabled.

Anti Reverse Engineering

An attacker who wishes to examine the binary code may try to attach it a debugger or send a SIGSEGV signal to the process to generate a code dump.

Attaching a debugger to the processes will not make the decrypted code observable because the decrypted function is executed in EL2 and as such cannot be accessed by the debugger.
If the attacker will try to modify the padded function code by setting a breakpoint inside it this breakpoint will never be reached but will overrun by Hypervisor when decrypting the function.
In general, it is not possible to debug TrulyProtect ARM programs because if a breakpoint is hit it would generate an exception to the Hypervisor; The hypervisor will set the program's next instruction to be executed as the first instruction after the breakpoint and the program would run without stopping.

If an attacker would try to disassemble the function he will see only the padded code. This is because the program is never decrypted when it does not run.

If an attacker would try to core-dump the program by sending SIGSEGV signal then the signal will never reach the program when it runs the decrypted function. The program runs in EL2 outside the operating system and is not aware of any software signals. Only when the program exists EL2 it would process the SIGSEGV signal, but as shown previously in EL0 the program is always in its encrypted form.

Keys Protection

To protect the keys from an attacker we put the keys in registers that are accessible only from EL2 or EL3.If an attacker wishes to read and EL2 register he must access EL2 exception level. ARM architecture forbids direct access to a higher privilege level, for instance; trying to read or write to EL2 registers when executing in EL1 is an access violation. Also, TrulyProtect does not allow any code to replace its exception vector once it is set.

The keys are expected to be available to the Hypervisor when an encrypted program runs. The origin of the keys is beyond the scope of this document.

Processor State

In a multiprocessors computer, a processor may move to a so-called deep sleep state by dropping to privilege level EL3. Unlike WFE/WFI this actually removes the processor from the operating system. When the operating system kernel decides to boot the processor back then it is important to set back VBAR_EL2 to Truly's Hypervisor

Miscellaneous

Printing in EL2

Any function that needs to be executed in EL2 privilege level must be mapped prematurely to the Hypervisor translation table. Many functions also use global variables so these variables must be mapped also. There are also variables which are dynamically allocated mapped and unmapped. To make the hypervisor accessible to generic routines like printk is too intrusive to the kernel code. For this reason, we've decided to implement el2_sprintf that would be executed in EL2 context but the actual printing would be performed in EL1 after exiting from EL2. Also, since sprintf format uses constant globals such as “foo foo %d” it means that all the read-only segment in the kernel must be mapped to the hypservisor. Due to this overhead, it is up to the programmer to choose whether he wants to el2_sprintf or not.

Stack protector

A stack protector is a gcc option that checks for stack corruptions. It adds a guard variable to a function. When a function is entered then the guard is initialized to some variable and when the function exists the guard is checked again. If the check fails then an error message is printed.
The Linux kernel provides a stack protector configuration option. When running a code in EL2 that executes function which is protected with a stack protector it would call functions that are not mapped to the Hypervisor. When this happens, the execution will fail. Failing when decrypting will crash the entire operating system.

For this reason, it is not possible to compile the kernel with a stack protector.

What Next

Hardware

It is not enough to boot the Linux Kernel on an ARMv8a to have an active Hypervisor. This is because the boot loader needs to run the kernel into EL2 [10] and the kernel needs to fill VBAR_EL2 with an initial exception vector.
At the moment Truly's technology runs on Hikey[9] because Hikey’s bootloader was modified to boot into EL2. Qualcomm Snapdragon SOMs do not boot into EL2 but into EL1.It is possible to set an hypervisor by accessing Qualcomm’s QSEE[13] but this requires authentication from Qualcomm.

Next Features

Feature	Description
Virtual Interrupts	It is essential to add support to VGIC.
Threads	Multi-threaded programs are not yet supported
32 bit	32bit Programs over 64bit kernel are not yet supported.
Thumb mode	Thumb mode instructions are not yet supported
Programming Languages	Other programming languages than c and c++ such as Java are not yet supported,
SMP	SMP is not yet fully supported. If a process migrates from one processor to another then at the time of this writing it would probably crash or hang.

BENCHMARKS

We test the the IPA and TEE. The tests are conducted in a Hikey board [6] . A Hikey is a small system on a chip ARM based computer manufactured by LeMaker. It is ARM8a the boots the kernel into EL2 and can run kvm on ARM.

SoC	HiSilicon Kirin 620
CPU	ARM8a Cortex-A53
CORES	8 Cores
Frequency	1.2 GHz
RAM	2GB
RAP-TYPE	LPDDR3 1.6 GHz

The software used is:

Linux Kernel	4.4.11
Distribution	Debian
compiler	gcc-linaro-4.9-2015.02-3-x86_64_aarch64-linux-gnu

IPA

This test measures the overhead of a two stage translation versys a one stage translation. The hardware used is Hikey and the test software is ramspeed. The tests were conducted by two kernel with the same configuration.

4.4.11 without truly
4.4.11 + truly patches

Single stage translation

```
INTEGER   Copy:    2734.28 MB/s
INTEGER   Scale:   1556.19 MB/s
INTEGER   Add:     2604.85 MB/s
INTEGER   Triad:   1471.42 MB/s
---
INTEGER   AVERAGE:   2091.69 MB/s
```

Two stage translation

```
INTEGER   Copy:    2676.92 MB/s
INTEGER   Scale:   1524.71 MB/s
INTEGER   Add:     2536.69 MB/s
INTEGER   Triad:   1441.91 MB/s
INTEGER   AVERAGE:   2045.06 MB/s
```

Conclusion

It is easy to see that there is an overhead of 2% in favor of a single stage translation.

Trusted Execution

In theses tests we want to check the performance degradation of executing in TEE.

Benchmark #1 : Execution and Decryption

The test is an FFT code (Fast Fourier function). We chose this routine because it traps to EL2 when first entered and then it traps to EL0 when it reaches its end. There are no traps from EL2 to EL2 in this test.
We devised two distinct versions of the hypervisor, the first one (4.4.11-rc1) decrypts in real time and the second (4.4.11-rc4) caches the decryption into a buffer.

The test program runs the FFT function a N times. The values are in microseconds.

Iterations	Reference	4.4.11-rc1	4.4.11-rc4
100	87	33124	1789
1000	873	331056	17976

We can see that constantly decrypting and flushing the cache is by a factor of 380. If we cache the the decrypted code and copy it each time by 20. This is a good improvement but not enough.

Appendix

Appendix A

```
void fft(float input[16],float output[16] )
{
  float temp, out0, out1, out2, out3, out4, out5, out6, out7, out8;
  float out9,out10,out11,out12,out13,out14,out15;

  out0=input[0]+input[8];
  out1=input[1]+input[9];
  out2=input[2]+input[10];
  out3=input[3]+input[11];
  out4=input[4]+input[12];
  out5=input[5]+input[13];
  out6=input[6]+input[14];
  out7=input[7]+input[15];
  out8=input[0]-input[8];
  out9=input[1]-input[9];
  out10=input[2]-input[10];
  out11=input[3]-input[11];
  out12=input[12]-input[4];
  out13=input[13]-input[5];
  out14=input[14]-input[6];
  out15=input[15]-input[7];
  temp=(out13-out9)*(SIN_2PI_16);
  out9=out9*(C_P_S_2PI_16)+temp;
  out13=out13*(C_M_S_2PI_16)+temp;
  out14*=(SIN_4PI_16);
  out10*=(SIN_4PI_16);
  out14=out14-out10;
  out10=out14+out10+out10;
  temp=(out15-out11)*(SIN_6PI_16);
  out11=out11*(C_P_S_6PI_16)+temp;
  out15=out15*(C_M_S_6PI_16)+temp;
   out8+=out10;
  out10=out8-out10-out10;
  out12+=out14;
  out14=out12-out14-out14;
  out9+=out11;
  out11=out9-out11-out11;
  out13+=out15;
  out15=out13-out15-out15;
  output[1]=out8+out9;
  output[7]=out8-out9;
  output[9]=out12+out13;
  output[15]=out13-out12;
  output[5]=out10+out15;
  output[13]=out14-out11;
  output[3]=out10-out15;
  output[11]=-out14-out11;
  out0=out0+out4;
  out4=out0-out4-out4;
  out1=out1+out5;
  out5=out1-out5-out5;
  out2+=out6;
  out6=out2-out6-out6;
  out3+=out7;
  out7=out3-out7-out7;
  output[0]=out0+out2;
  output[4]=out0-out2;
  out1+=out3;
  output[12]=out3+out3-out1;
  output[0]+=out1;
  output[8]=output[0]-out1-out1;
  out5*=SIN_4PI_16;
  out7*=SIN_4PI_16;
  out5=out5-out7;
  out7=out5+out7+out7;
  output[14]=out6-out7;
  output[2]=out5+out4;
  output[6]=out4-out5;
  output[10]=-out7-out6;
}
```

References

[1] http://www.trulyprotect.com

[2]https://docs.microsoft.com/en-us/windows/device-security/device-guard/device-guard-deployment-guide

[3] https://software.intel.com/en-us/sgx

[4] ARM InfoCenter. http://infocenter.arm.com/help/index.jsp

[5] Blue Pill Software: https://en.wikipedia.org/wiki/Blue_Pill_(software)

[6] ARM Architecture Reference Manual. ARMv8, for ARMv8-A architecture profile . 2013-2016

[7] ARM Architecture Reference Manual Supplement . ARMv8.1, for ARMv8-A architecture profile. Errata markup. 2016

[8] Understanding the Linux Kernel. O’reilly Daniel P Bovet, Marco Cesati. 3-rd edition. 2005

[9] http://www.96boards.org/product/hikey/

[10] Supporting KVM on the ARM architecture; lwn 2013. https://lwn.net/Articles/557132

[11] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho and Sarah Martin Arizona State University . TrustZone Explained: Architectural Features and Use Cases. 2016 IEEE 2nd International Conference on Collaboration and Internet Computing

[12] Jeremy Andrus, Christoffer Dall, Alexander Van’t Hof, Oren Laadan, and Jason Nieh. Cells: A Virtual Mobile Smartphone Architecture.

[13] Qualcomm Secure boot and image authentication. secure-boot-and-image-authentication-technical-overview.pdf . Ryan P Nakamoto Staff Engineer , Qualcomm Techonologies