International Journal of Information Security Special Issue on "IoT Security and Privacy" manuscript No.
(IJIS-D-18-00538R2)

Protection against reverse engineering in ARM

Raz Ben Yehuda[a,1,3], **Nezer Jacob Zaidenberg**[b,1,2,3]

[1]University of Jyväskylä, Jyväskylä, Finland
[2]College of Management Academic Studies, Street, Rishon LeZion, Israel
[3]TrulyProtect, Finland

Received: 09 Nov 2018/ Accepted: 18 Jun 2019

Abstract With the advent of the mobile industry, we face new security challenges. ARM architecture is deployed in most mobile phones, homeland security, IoT, autonomous cars and other industries, providing a hypervisor API (via virtualization extension technology). To research the applicability of this virtualization technology for security in this platform is an interesting endeavor. The hypervisor API is an addition available for some ARMv7-a and is available with any ARMv8-a processor. Some ARM platforms also offer TrustZone, which is a separate exception level designed for trusted computing. However, TrustZone may not be available to engineers as some vendors lock it. We present a method of applying a thin hypervisor technology as a generic security solution for the most common operating system on the ARM architecture. Furthermore, we discuss implementation alternatives and differences, especially in comparison with the Intel architecture and hypervisor with TrustZone approaches. We provide performance benchmarks for using hypervisors for reverse engineering protection.

Keywords Security · ARM · Mobile · IoT · Hypervisor

1 Introduction

We explore Man-at-the-Endpoint (MATE) attacks on ARM-based systems such as embedded and mobile devices and focus on protection against reverse engineering for the ARM platform.

The TrulyProtect [3] microrvisor is a secured thin hypervisor based on the blue pill concept [8] for the x86 environment. TrulyProtect does not run on multiple operating systems and is used only for security purposes. TrulyProtect was initially implemented and benchmarked on an x86 CPU architecture. We examine the necessary modifications for porting TrulyProtect to ARM, its security benefits, and performance costs. TrulyProtect provides a thin layer of code that is invoked through traps on the x86 architecture. We use a similar approach using exceptions to elevate from lower privilege levels in ARM. These traps can be generated from userspace programs (EL0) or kernel code (EL1).
In general, our technique is composed of the following steps:

[a]e-mail: raz@trulyprotect.com
[b]e-mail: nezer@trulyprotect.com

- Static phase
  Encrypt some segment of the ELF binary.
  Replace this segment with a trap opcode.
- Runtime phase
  Whenever the processor executes the trap opcode, the processor moves from user mode to HYP mode, decrypts the encrypted code, executes it in HYP mode, encrypts it back and returns to user mode.

In this paper, we examine the anti-reverse engineering of native code. We present a thin hypervisor implemented in an ARMv8-a 64-bit processor, and synthetic benchmarks. The hypervisor can be considered a Trusted Execution Environment (TEE) as it offers an isolated execution environment for decrypted code.

2 Background

We present the ARM architecture permission model and past work on TrulyProtect under an Intel platform.

2.1 ARM permission model

The ARMv8-a platform normally has four exception (permission) levels:

Exception level 0 (EL0) refers to normal userspace code. EL0 is analogous to "ring 3" in the x86 platform. For example, virtually all applications and games on a standard iPhone or Android phone run on EL0.

Exception level 1 (EL1) refers to operating system code. EL1 is analogous to "ring 0" in the x86 platform. For example, the Android or the iOS (the operating system itself) on a mobile phone runs on EL1.

Exception level 2 (EL2) refers to HYP mode. EL2 is analogous to "ring -1" or "hypervisor mode" on the x86 platform. In most ARM devices, nothing runs on this exception level unless the ARM device starts a hypervisor when it boots.

Exception level 3 (EL3) refers to TrustZone™.
TrustZone is a special security mode that can monitor the ARM CPU as well as the operating system that it runs. TrustZone allows running a separate security real-time operating system in a secure world. There are no directly analogous modes, but similar concepts in x86 (from security perspectives) are Intel's ME [9] and SMM [10].
Each of the exception levels provides its own state of registers and can access the registers that correspond to the lower permission levels, but not the higher levels. In Figure 1, Exception Levels EL3, EL2 and EL0/EL1 have their own translation tables. Thus, moving between exception levels requires a change of the entire address space.

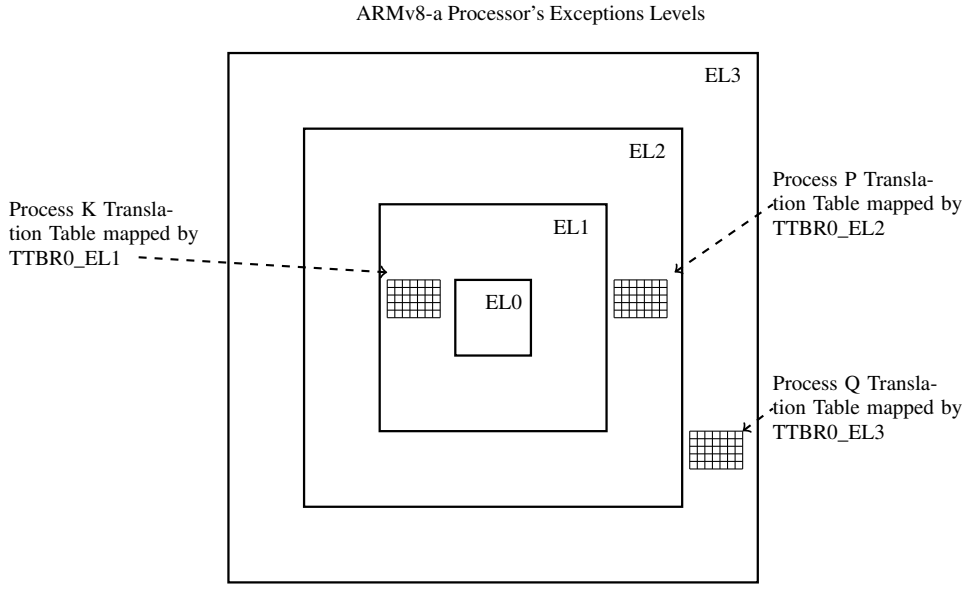


Fig. 1 MMUs separation

2.2 TrustZone

EL3 refers to the TrustZone architecture [11, 12]. TrustZone is a special ARM security mode that allows the running of a "secure OS" in parallel to the normal "insecure OS." The normal (insecure OS), called "normal world," is the standard operating system that normally runs on the device (for example, VxWorks, Linux, iOS or Android). The secure OS, called "secure world," is an implementation-specific real-time operating system (such as OKL4[13]) and is chosen by the hardware vendor. The purpose of the secure world is to attest the normal world and provide a root of trust and trusted computing services. The secure world is separated from the non-secure world by the hardware memory protection unit.

Users cannot normally run applications in the secure world. Furthermore, running applications on TrustZone is usually prevented by the vendors.

ARM Holdings introduced TrustZone as part of its architecture in 2009. TrustZone was an optional extension to the ARMv7-a architecture. TrustZone is part of the ARMv8-a architecture and described in detail by Ngabonziza [14]. The TrustZone can be used in two distinct operational modes:

1. Monitor mode provides a separate operating system to run concurrently with a second, generic operating system (such as Linux). The ARM processor itself assigns resources (such as processing cycles) to the TrustZone real-time operating system periodically.
2. Passive library mode includes a monitor exception vector that is activated through traps or SMC calls.

It might have been possible to implement TrustZone as a virtual machine running on EL2. However, TrustZone is designed to execute in an isolated environment and at a higher privilege level than the hosting machine and hypervisor.

Unfortunately, a malicious application running in TrustZone or HYP mode may be used to attack the TrustZone operating system or the hypervisor itself. However, these attacks

are only possible if the malicious application runs in EL2 or EL3. Fortunately, installing malicious input can be detected by TrustZone in the Static Root of Trust Measurements (SRTM) process.

2.3 TrulyProtect

On x86 platforms TrulyProtect provides anti-reverse engineering [15], endpoint security [16], video decoding [25], forensics etc. TrulyProtect relies on Dynamic Root of Trust Measurement (DRTM) attestation to create a trusted environment in the hypervisor and receive encryption keys [17].

After receiving the keys, TrulyProtect provides anti-reverse engineering by executing encrypted code in the hypervisor and protecting the decryption keys. In this work, we review the performance and capability of similar hypervisor operations on the ARM architecture. Additional work includes endpoint security by controlling instructions (memory pages) that are allowed on an endpoint. The hypervisor reads an encrypted and signed database of allowed pages. The hypervisor grants execution permission only to pages that are allowed to execute on the endpoint. Platform independent extensions to the TrulyProtect hypervisor also allow protection against reverse engineering of managed code and protection of encrypted video. These features could be ported to the ARM architecture as well but were beyond our scope.

2.4 Attestation

TrulyProtect DRTM (Dynamic Root of Trust Measurements) relies on an attestation method based on a technique similar to Kennel and Jamieson [27,28]. This attestation method is not required on ARM (though we believe similar methods can be developed); thus, the Static Root of Trust Measurements (SRTM) using TrustZone is used instead. If TrustZone is unavailable, using TPM [19] and Secureboot is another attestation option [15]. If the hardware root of trust is not available, then even DRTM [29] is possible. Regardless, we provide the decryption keys only after successful attestation. We have no innovation in this field and thus, attestation remains out-of-scope.

2.5 Protecting the hypervisor

In embedded devices, such as mobile phones or ARM servers used for cloud computing, a malicious user may try to compromise the computer‘s [30] hypervisor [31,32] as also described in [33] for Xen or KVM [35] through a host privilege escalation. Min [34] claims that the best approach is to rely on the hardware and presents a security monitor as a solution. We agree with this approach and offer to do the attestation in, for instance, the TrustZone. However, we examine what possible vulnerabilities a malicious software might try to take advantage of:

- A boot loader or kernel replaced. The chain of trust would prevent the bootloader or the kernel from executing.
- A malicious program. A TrulyProtect’s signed program with a malicious code generates the escape sequence to enter the VM. This means that part of the code tries to access privileged registers. We trap any access to these registers while executing in EL2 so any

attempt would be trapped to the hypervisor. For example, we disable the SMC calls and HVC calls while executing in EL2. In cases when there is no protection in EL2, we can protect from the EL3 (TrustZone).
– A malicious hypervisor. It is not possible to replace the TrulyProtect hypervisor once it is set.

Once a trustworthy hypervisor is running, it can protect the keys.

3 Related works

Many researched techniques for trusted execution, in this section, we describe some.

3.1 TEE in ARM

Ekberg et al [43] discuss TEE in mobile devices as having strict requirements that date back to the beginning of the mobile device industry and describes the various techniques of protection, a chain of trust, trusted storage, and others. They also mention virtualization as a means of protection through the use of a VMM (Virtual Machine Monitor). These VMMs run several concurrent guests isolated one from the other. The TrulyProtect thin hypervisor does not rely on a VMM and is not considered a guest.

3.2 Code obfuscation and DRM

A program can be obfuscated to diminish the chances of successful reverse engineering and discovery of its trade secrets, modification, etc. There has been significant work on systems to obfuscate and de-obfuscate code. Vot4CS [46] is a relatively recent obfuscation for C# that survives many de-obfuscation attempts. Kevin Coogan et al [7] and Kalysch [42] describes means to attack such obfuscators. Szor [36] discusses automatic de-obfuscation in order to detect computer viruses. Such methods allow code to execute on a target machine and make reverse engineering much more challenging. However, code obfuscation is bound to fail eventually [47]. In practice, experience shows that by investing time and dedication, these methods can frequently be broken faster than people think. For example, the Nintendo Wii-U DRM system is notorious for being broken less than one month after the platform was released despite Nintendo having full control on the hardware operating system and the software [38].

DRM [17] techniques require protection against reversing as they are a frequent target of reversing and removal attempts. TEE and reverse engineering technologies can be used on many entertainment systems for DRM protection. Devices such as smart TVs, handheld devices, TV set-top boxes and game consoles [6] are all examples of using modern hardware for these purposes. Hardware modern security features are used to prevent copying whenever such features are available.

3.3 Intel SGX

Intel SGX [1] is a set of instructions added to the processor that enables the use of protected and isolated memory regions knows as "enclaves." Access to such enclaves requires special software tools and expertise.

3.4 OKL4 Microvisor

OKL4 Microvisor [18] is a secure hypervisor supported by Cog Systems. The OKL4 microvisor supports both para-virtualization and pure virtualization. It is designed for the IoT industry, and supports ARMv6, ARMv5, ARMv7-ve [4] and ARMv8-a [5]. Unlike TrulyProtect, the OKL4 microvisor is a full kernel executing in HYP mode. Cog Systems also offers an SDK called "D4 secure SDK" and an RTOS.

4 Anti-reverse engineering

TrulyProtect [3] for the ARM thin hypervisor offers an easy way to execute code in a secure environment in ARMv8-a, and does so in a way that does not require the user to modify the code. Unlike QSEE [20,21], the interaction with the secured area does not require any special preparations. TrulyProtect is a real thin hypervisor, i.e., its footprint is less than 100kB when counting the AES decryption. It does not offer a system-wide solution but focuses on protecting distinct parts of the program. Memory protection, anti-reverse engineering and protection of keys are all implemented like other platforms supported by TrulyProtect.

5 Innovation

In this section, we detail the proposed system for anti-reverse engineering in ARM.

5.1 Program protection in ARM

We now examine the anti-reverse engineering process. Anti-reverse engineering is often achieved using obfuscation; however, here, we want to prevent the reverse engineering of software by encrypting the software code before deployment and deploying only the encrypted software. We make the following assumptions:

- The encryption function we use is safe and cannot be broken. We use AES [22]; Nevertheless, if in the future, AES is broken [23] then the encryption function can easily be replaced with an elliptic curve [24] or any other encryption function. (We do not assume anything about the encryption function.)
- The CPU itself is sufficiently complex to prevent the attacker from "looking" inside the CPU.
- We assume the CPU works according to the specifications and no hidden modes allow internal CPU structures to be read.

5.2 The proposed system

Under the above assumptions, we provide evidence that the decrypted software is not available to the normal operating system, and the hypervisor will undertake the protection for the software and the decryption keys. This proposed system is composed of two phases.

1. Static encryption
   We choose which functions we wish to encrypt and obtain a binary copy of the program. Then we use TrulyProtect's instrumentation tool to encrypt the chosen functions.

2. Runtime execution
   The program runs as-is. Each time the processes access the encrypted function, the processor drops to HYP mode, decrypts the function, and executes it.

5.2.1 Static encryption

As noted earlier, a program protection framework in ARM mimics the way TrulyProtect protects programs in x86. However, because the ARM hypervisor differs greatly from other hypervisors (such as the x86 hypervisor), we will describe how it is being undertaken in detail.
Figure 2 depicts the first stage of the encryption of a single function. An ELF ARM binary is processed, and the instructions of the function foo() are replaced with a trap code, the "BRK" instruction.

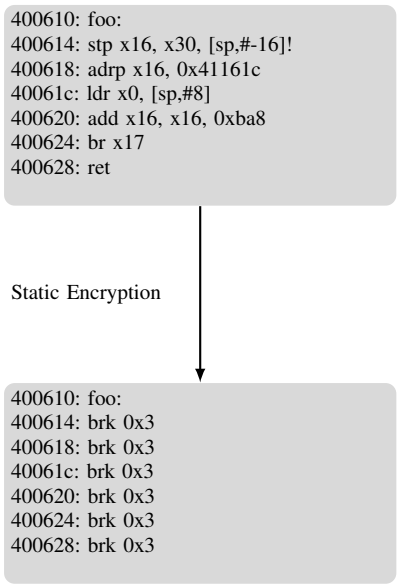


Fig. 2 Static Encryption - replace a function

We chose the "BRK" instruction for two reasons:

1. It easy to configure to trap into the HYP mode by setting the mdcr_el2 register.
2. It does not change any value of the general purpose registers; thus, when trapping to the hypervisor, the program's context can be saved.

The addresses from the left are relocatable, meaning that the actual addresses are not known at the static encryption phase. It is important to note that the ARMv8-a exception level model dictates that only positive addresses, i.e., userspace addresses can be executed in EL2, and not negative (kernel addresses).
Static encryption also includes the addition of the new encrypted function foo(). The encrypted version of the function is added to the ELF binary as a new segment, as depicted in Figure 3.

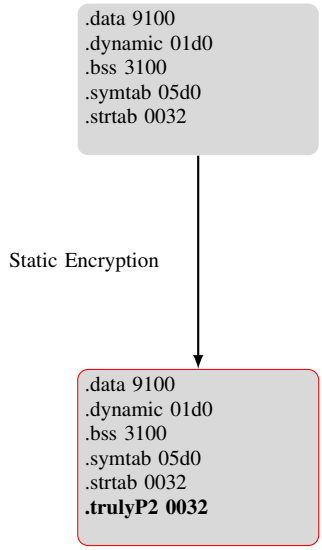


Fig. 3 Static Encryption - additional segment

5.2.2 Runtime decryption

The next phase is when the program is loaded and executed. To understand this phase we need to explain ARM's memory model. But first, we start with the x86 memory model. In x86 accessing a process or a kernel's virtual memory from HYP mode does not require mapping (Figure 4).

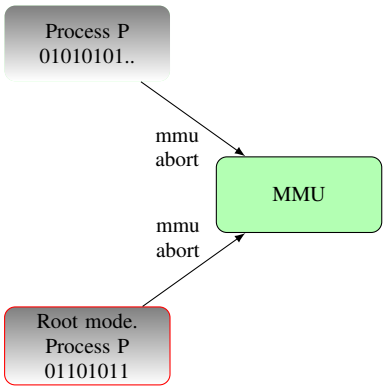


Fig. 4 x86 model. A single translation table

However, in ARM it is required to map the designated pages to the hypervisor translation table. For this reason, when we want to access EL0 code or data from EL2, we must first map the pages to the hypervisor. As a result, a page is mapped twice: one to each of two distinct memory tables of the same process, as shown in Figure 5.

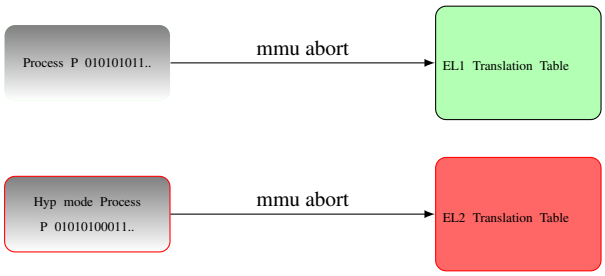
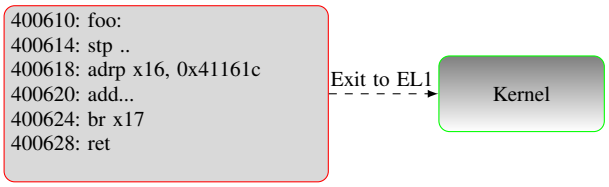


Fig. 5 ARM model. Translation tables are not shared

Thus, when any arbitrary program executes, the kernel searches in the ELF segments for the segment .trulyP2, and if it is found, then the kernel calls the hypervisor to enable trapping the "BRK" instruction. As long this process runs, any time the processor executes "BRK", it will trap into HYP mode. The hypervisor verifies that the current position of the instruction pointer is in foo(), and if so, it decrypts the encrypted version of foo() accommodated in .trulyP2 onto its original position. Then it starts executing from where the trap was and continues to execute in EL2 until one of the followings occurs:

1. foo() reaches its end, performs the ret instruction, and returns to the hypervisor. The hypervisor flushes the caches and TLBs, put back the trap code and return to EL0.
2. foo() accesses unmapped memory which results in an EL2 MMU abort (Figure 6.)
3. foo() performs an svc (a system call).

The ARM memory model poses a problem in MMU aborts. If there is an MMU data abort in EL2, then the unmapped region must also be mapped to EL0 memory page tables if it was not already mapped before. For this reason, we only map the trap-code (the code that generates the exceptions), the encrypted code, and the stack. This way, we know that the MMU aborts because of an unmapped region in EL2 while the region is mapped in EL1/EL0 (or will be mapped). As we show later, this approach has a severe performance penalty. For this reason, we tried a different approach - a real-time mapping, referred to here as **Rt-map**.

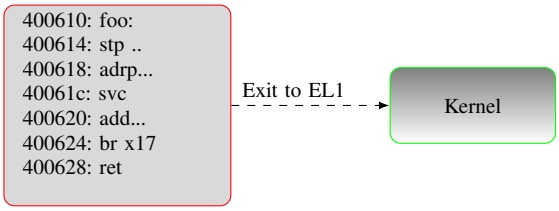


Instruction 400618 generated an MMU abort because address 0x41161c was not mapped to EL2

Fig. 6 Real-time mapping in EL2

In Rt-map mode, when an MMU abort takes place in EL2, we exit to EL1 and access the faulting address to map it to EL0/EL1 (assuming it was not already mapped); then, we map

the faulted address to EL2 and continue execution in EL0. With this approach, the next time this code is accessed in EL2, it will not MMU-abort in EL2 as previously but will continue to execute until the next MMU abort. This way, most of the process's address space gradually maps to EL2 as the process executes. Consequently, the process exits to EL0 (and then EL1) only when it performs a system call (Figure 7). Thus, an interposition is kept between the EL2 and EL0 processes.



The SVC is designed to exit from EL2 to EL0. The program counter is programmed to re-execute the SVC in EL0 again so that the SVC trap would be generated from EL0 to EL1

Fig. 7 System call in EL2

This way, the decrypted content is never available to the operating system. If a user tries to dump the memory image of the process while it is in EL2, then the processor would leave HYP mode and, consequently, put back the trap-code.

5.3 The hyplet

Protecting EL1

The hyplet is a generic term we use to describe programs that partly execute in EL0 or EL1 and partly in EL2. The technique we introduce in this paper is a special case of an encrypted hyplet. The hyplet as a general term is not part of the paper, but we provide some details for how it works. It is a challenging task to protect device drivers in ARM8v-a. For this reason, we developed the hyplet ISR (Interrupt Service Routine), in short hypISR. The hypISR is a means to move from EL1 to the userspace program without a noticeable penalty. A hyplet-ed program is a program that constantly accommodates EL2 translation table, and is not evacuated from EL2's MMU.

Whenever an interrupt is being processed by the processor; if the data or code need to be protected, then the processor moves to EL2 and from there to the userspace program that handles it. The callback function that processes this interrupt routine is the hyplet. Through this dedicated userspace process, sensitive data can be protected by reading it into a protected memory region.

To ensure that both the program code and data are always accessible, it is essential to disable evacuation of the program's translation table from the processor. Therefore, we chose to constantly accommodate the code and data in the hypervisor translation registers (Figure 8) [4]. To map a userspace program, we modified the Linux ARM-KVM mapping infrastructure to map userspace code with kernel-space data [37].

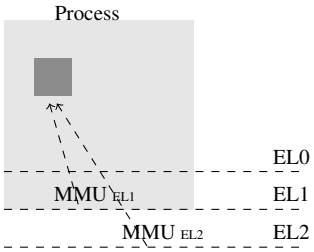


Fig. 8 Asymmetric dual view

Figure 8 shows identical addresses mapped to different virtual addresses in two separate exception levels. The small square subsection is mapped to EL2 and is therefore accessible from EL2. Usually, when executing in EL2, EL1 data is not accessible without premature mapping to EL2. This mapping extends the ability of a Linux process to execute from two exception levels to three.

Protecting the hypervisor MMU

In the standard method for memory mapping, EL1 is responsible for EL1/EL0 memory tables, and EL2 is responsible for its own memory table, in the sense that each privileged exception level accesses and manages its own memory tables (Figure 9).

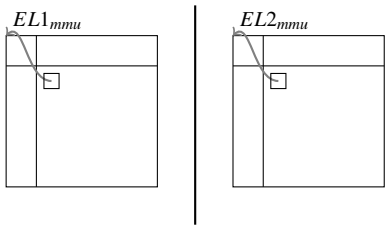


Fig. 9 Memory table access

This approach, however, puts the microvisor at risk because it might overwrite or otherwise garble its own page tables. As noted earlier, ARM8v-a hypervisor has a single memory address space (unlike TrustZone that has two, for the kernel and the userspace). The ARM architecture does not coerce an exception level to control and access its own memory tables, allowing the ARM architecture to map the EL2 page table in EL1 (Figure 10). As such, only EL1 can manipulate the microvisor page tables.

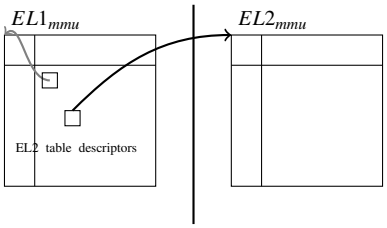


Fig. 10 Memory table access for hyplets

5.4 OS dependence

The ARM hypervisor is designed to be distinct in execution and memory context from other exception levels. As a result, whenever the hypervisor needs to access EL0 pages, it must map them to its own translation table. However, for that to happen, the hypervisor must implement its own page allocation system because there might be a need to allocate a table. However, we refrain from accessing the translation tables of EL2 in EL2 to reduce risks. Also, it is better to re-use the KVM-ARM [37] allocation system because it is a mature software. For these reasons, we decided to use KVM-ARM; thus, the hypervisor in this paper refers only to Linux implementations.

6 Evaluation

In the following sections, we estimate our microvisor overhead. we evaluate IPA overhead and TrulyProtect technology to TrustZone based solutions and measure encrypted program execution overhead. We measure the penalty of the repeated encryption, cache and TLB evacuations that are caused by system calls or minor page faults in EL2. We test the microvisor in typical I/O operations and under CPU loads. We also describe the means to mitigate these penalties.
The measures presented in the tests are averages, standard deviations, minimum and maximum. Each experiment was performed five times.
The tests were conducted in a Lenovator “Hikey” board. A Hikey is a small system-on-a-chip ARM-based computer manufactured by LeMaker. Hikey’s processor is an ARMv8-a.

SoC	HiSilicon Kirin 620
Number of CORES	8
Frequency	1.2 GHz
RAM	2GB
RAP-TYPE	LPDDR3 1.6 GHz

Table 1 Test Hardware

The software used was:

Linux Kernel Version	4.4.11
Distribution	Debian
Compiler	gcc-linaro-4.9-2015.02

Table 2 Test Software

6.1 IPA overhead

We wish to evaluate the cost of using the Intermediate Physical Address (IPA) [39]. The IPA is a second stage of translation and is used to separate the guest operating system from the physical memory by a double memory fault mechanism.

We measured the overhead of a two-stage translation compared with a single stage translation. The test software was RAMspeed.

The tests were conducted by two kernels with the same configuration.

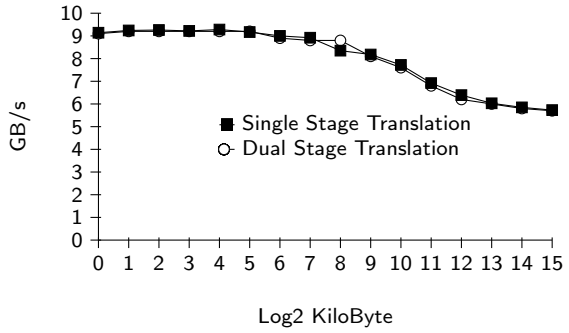


Fig. 11 IPA vs Native, WRITE access

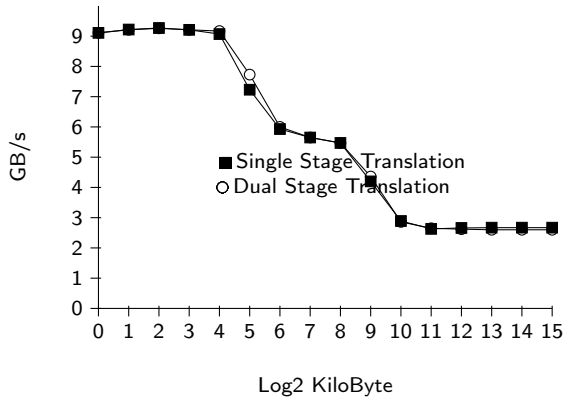


Fig. 12 IPA vs Native, READ access

In Figure 4 we attempt to simulate more closely the real-world computing load. A, B, and C are locations in the memory; M in SCALE is a constant.

SCALE	A = m*B
ADD	A + B = C
COPY	A=B

Table 3 IPA Tests explained

Single stage	COPY	SCALE	ADD
Avg	2.03	1.60	1.65
StdDev	0.01	.0.1	0.008
Max	2.05	1.62	1.66
Min	2.02	1.59	1.64

Dual stage	COPY	SCALE	ADD
Avg	2.08	1.62	1.69
StdDev	0.01	0.005	0.008
Max	2.1	1.63	1.7
Min	2.08	1.62	1.68

Table 4 Real Load GB/s

Evidently, there is no difference between using a two-stage translation and a single stage translation. We have shown that IPA does not influence the memory access performance.

6.2 TEE enter/exit overhead

The common way to enter TrustZone is when the processor executes the SMC call from EL1. In TrulyProtect, the processor traps to EL2 when it executes the "BRK" instruction. TrustZone implementations include a kernel space driver and the data are passed through ioctl(2)s operations that use SMC. In this test, we evaluate how costly the "BRK" trap is, when it traps to EL2, compared with the SVC/HVC (SVC is a system call to EL1 and HVC is a hypervisor call to EL2) and ioctl (2) [41] system call. We assume the HVC overhead is similar to SMC.

Trap	BRK trap	Pure SVC call	Ioctl syscall
Avg	92 ns	92 ns	490 ns
StdDev	41 ns	41 ns	65 ns
Max	156 ns	156 ns	550 ns
Min	52 ns	52 ns	440 ns

Table 5 BRK vs. SVC & Ioctl

There is a very little difference when comparing a trap to a typical SVC call. Executing ioctl that does nothing costs on average about 500ns without pre-emption. This is because there are many operations undertaken by the kernel before it returns. Since executing a

"BRK" trap takes on average 100ns and an ioctl takes 500 ns, we can say that the TrulyProtect mechanism for entering and exiting the TEE is five times faster than TrustZone-based technologies.

6.3 Encrypted code CPU use

We have used an FFT (Fast Fourier Transformation) program as a benchmark. The FFT program is a CPU-intensive program. This program does not perform any IO-related tasks. We chose this program because it traps to EL2 when it first enters the encrypted function and then continues to run until it reaches its end, and returns to TrulyProtect's hypervisor. There are no traps from EL2 to EL2 in this test. The following presents several decryption strategies.

- **Native** A clear-text run.
- **Live-decrypt** Decrypts in real time and flushes the instruction cache on exit from HYP mode. When the hypervisor is entered again, it decrypts again.
- **Cache** The first time the hypervisor enters the encrypted code, it caches the decrypted code into a temporary protected buffer and from now on the buffer is copied onto the trap-code each time the hypervisor is entered for execution. On exit, the function's decrypted instructions are flushed from the L1 instruction cache. (non-transient write-back).
- **RT-map** In addition to caching the decrypted data, i.e.; the "cache" mode, the hypervisor maps other parts of the process's address space into the hypervisor in real time to reduce exits from HYP mode. Reducing the exits reduces the cache flushes and copying.

6.3.1 First execution overhead

We measured the duration of an encrypted function when it first entered. The overhead includes the context switch to hypervisor mode and the time required for the decryption.

The first execution of an encrypted code segment bears the penalty of the decryption; therefore, we assume the performance penalty of running encrypted code will be larger. Table 6 shows the duration of the first and single call to FFT in each of the configurations aforementioned.

	Avg	StdDev	Max	Min
Native	4.6	0.54	5	4
Live-decryptc	460	27	480	421
Cache	407	28	456	385
RT-map	476	14	485	451

Table 6 Duration of a single FFT in microseconds

As can be seen, the first call to FFT is time-consuming. In the best case, it is 80 times worse than the reference test, which is five microseconds.

6.3.2 Repeated execution overhead

We measured the time for additional executions of encrypted code beyond the first. In the cached and real-time mapping modes, after the first run, the overhead with additional calls includes the overhead of the context switch to HYP mode but does not include the time required for the decryption, as this work was already completed earlier.

In Table 7 we executed the FFT after the function was decrypted into a temporary buffer (in the cache and RT-map cases). We benchmarked 1,000 consecutive calls to the FFT function.

	Avg	StdDev	Max	Min
Native	888	5	896	884
Live-decrypt	398914	2717	402444	396012
Cache	7323	346	7712	6800
RT-map	928	3	934	926

Table 7 Duration of a 1000 FFT calls in microseconds

From Table 7, we see that the fastest mode is the real-time mapping, and has the smallest deviation. In the live-decrypt mode, the overhead is caused by the constant decryption, when entering the hypervisor, as well as putting back the pad code when exiting the hypervisor back to EL0 and, lastly, flushing the cache. In the cache mode, we can see how the overhead of the repeated decryption impacts the speed. In real-time mapping, we gradually map the process's address space to the hypervisor, so there is no exit from the hypervisor except when the process exits. Because the program is running without any interrupts and exits the hypervisor only a single time, we obtain an execution time close to the Native execution time. We can also see that in real-time mapping execution time is more predictable than the other alternatives.

We, therefore, conclude that it is best to remain in EL2 as much as possible. The cost of constantly decrypting and padding back is significant.

6.4 Predictability

To show the RT-map mode is faster than the Native mode because it runs without interrupts, we performed an additional test. Figure 13 is a CPU-intensive FFT function being executed in a tight loop. We generated a large number of interrupts (approximately 3,000 network interrupts/second, while in idle state is approximately 300 network interrupts/second) and then we executed a simple FFT function one thousand times. The interruptless mode is when we executed FFT through a hypervisor without any decryption. The reference test is when we executed FFT as a standard Linux function.

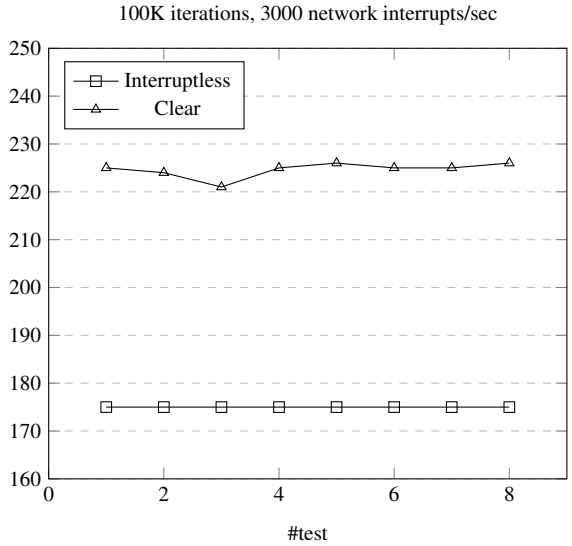


Fig. 13 FFT duration in milliseconds

Figure 13 shows a gap of over 28% between the two runs. We, therefore, conclude that the difference between RT-map to Native in Table 7 is due to the execution conditions.
Next, we want to measure more realistic loads. We measure malloc(3) [41] and free(3) [41], disk IO, file open(2) and close(2), memory access, and all these operations combined.
It is important to note that the proposed system supports the use of any of these operations (malloc(3), open(2), write(2)) in an encrypted context; however, it does not offer to obfuscate them. This system localizes the obfuscation to some functions in a program.
The next tests were performed only in RT-map mode.

6.5 Stack access overhead

In Table 8, we measure a real-life performance of memory access, read, and write when the memory is allocated on the stack. We encrypted a small function so the overhead of decryption and cache eviction would be low.

Iterations	Measure	Encrypted	Clear
1			
	Avg	325	25
	StdDev	43	0.9
	Max	378	26
	Min	280	24
10			
	Avg	328	28
	StdDev	23	1.7
	Max	365	30
	Min	304	26
100			
	Avg	388	58
	StdDev	34	1
	Max	423	60
	Min	351	57
1000			
	Avg	676	377
	StdDev	16	10
	Max	694	390
	Min	661	370

Table 8 Duration of stack access in microseconds

Stack access (Table 8) is an essential test because the stack is being mapped in real-time to the microvisor and the kernel (EL1). In this test, we used a stack of 10 pages. We wanted to evaluate how costly overhead of real-time mapping. As we can see, the first run is 13 times slower than the non-encrypted program with a 14% standard deviation. However, as the number of iterations grows, the overhead mitigates. In 1000 iterations it is 80% with 2% standard deviation. We, therefore, conclude it is best to use a pre-mapped memory, and if possible, pre-map the stack or any other memory that is accessed in the hypervisor.

6.6 A RAM access overhead

In Table 9, we access a heap memory randomly. We did not test the malloc(3) itself but only the memory access: a read and a write.

Iterations	Measure	Encrypted	Clear
1			
	Avg	369	163
	StdDev	27	0.7
	Max	402	164
	Min	342	162
10			
	Avg	1758	1566
	StdDev	28	6
	Max	1791	1578
	Min	1718	1562
100			
	Avg	15551	15481
	StdDev	24	21
	Max	15585	15513
	Min	15529	15459
1000			
	Avg	153461	154338
	StdDev	28	152
	Max	153502	154610
	Min	153439	154247

Table 9 Duration of RAM access in microseconds

In Table 9, we access a large amount of data (1MB), so the first MMU aborts duration is negligible compared to the memory access duration. When running 1000 iterations before exiting the hypervisor, the overhead of exiting the hypervisor is not noticeable.

Evidently, the less we exit the hypervisor, the lower the penalty. In general, it is best to map the heap memory to the hypervisor as early as possible, to reduce MMU aborts to EL2.

6.7 malloc(3)/free(3) overhead

We have benchmarked the standard memory allocator under Linux. In Table 10, we test the cost of malloc(3) and free(3) without accessing memory, i.e., we call malloc(3) and free(3) repeatedly.

Iterations	Measure	Encrypted	Clear
1			
	Avg	224	117
	StdDev	53	7
	Max	279	124
	Min	136	110
10			
	Avg	483	145
	StdDev	21	6
	Max	520	150
	Min	465	135
100			
	Avg	462	161
	StdDev	35	6
	Max	522	169
	Min	434	156
1000			
	Avg	734	409
	StdDev	14	15
	Max	752	430
	Min	713	395

Table 10 Duration of malloc(3)/free(3) access in microseconds

Here (Table 10), the overhead for a single iteration is 200% and is gradually reduced to 80% over 1000 iterations. This code does not perform any page faults as it does not access the allocated memory at all. Like in the FFT and the Stack access tests, we can see that the repeated decrypting, cache and TLB evacuation is approximately 80% for small functions.

In real-time sensitive programs, it is best to avoid malloc(3) and free(3) as much as possible. Because a CPU-bound program is unlikely to perform memory allocations in real time, this overhead can be avoided by using pre-allocation and prematurely mapping the RAM to the hypervisor.

6.8 A File open/close overhead

Table 11 presents measures of I/O performance associated only with opening and closing a file over Linux and the standard ext4 file system. The test opens and closes a single file 1,10...1000 times repeatedly.

Iterations	Measure	Encrypted	Clear
1			
	Avg	170	26
	StdDev	27	1.8
	Max	212	29
	Min	137	25
10			
	Avg	315	84
	StdDev	28	2.1
	Max	345	345
	Min	278	278
100			
	Avg	1070	632
	StdDev	15	7.5
	Max	1094	641
	Min	1057	623
1000			
	Avg	8768	5982
	StdDev	73	33
	Max	8890	6034
	Min	8697	5929

Table 11 Duration of open/close in microseconds

There is an overhead of 30% in favour of running the clear text in 1000 iterations. Like in the previous tests, the overhead decreases as the number of iterations grows, this is because for each open(2) and close(2) the hypervisor exits, and the duration of these system calls is small compared with the decryption of the test function.

In general, we can expect system calls to have some impact on performance due to context switches. We should try to decrease the code that produces system calls as much as possible. For instance, getting the time is extensively used in programs, so it is best to avoid getting the time through a system call but rather by accessing the timer clock register $cntvct\_el0$ directly.

6.9 A file write overhead

Table 12 measures IO performances associated with file writes under Linux and, the standard ext4 file system. In Table 12, we measure most of the above operations in addition to file writing operations. The test included memory allocation, file opening and closing, random memory allocation, memory access and memory freeing. The test was performed from a single file, up to 10 files.

#Files	Measure	Encrypted	Clear
1			
	Avg	3513	4497
	StdDev	417	158
	Max	4166	4769
	Min	3121	4395
2			
	Avg	9135	8494
	StdDev	418	200
	Max	9880	8849
	Min	8910	9381
3			
	Avg	11758	11876
	StdDev	1492	470
	Max	13577	12715
	Min	9885	11595
4			
	Avg	15724	15208
	StdDev	158	660
	Max	15965	16387
	Min	15552	14862
5			
	Avg	19130	18235
	StdDev	138	167
	Max	19280	18376
	Min	18983	17945
6			
	Avg	22678	21674
	StdDev	209	43
	Max	23006	21707
	Min	22488	21599
7			
	Avg	25913	25875
	StdDev	139	1378
	Max	26046	27419
	Min	25684	24766
8			
	Avg	29305	29220
	StdDev	146	1531
	Max	29475	30906
	Min	29142	28000
9			
	Avg	33273	32061
	StdDev	1253	1347
	Max	35453	34432
	Min	32410	31173
10			
	Avg	33156	34890
	StdDev	2402	214
	Max	38084	35099
	Min	30005	34563

Table 12 Duration of IO write in microseconds

It is noticeable that the more IO is processed, the less the difference between the two executions. In Table 12, the effect of running our security hypervisor is unnoticeable. Encryption works with a negligible overhead in most cases. The decryption and cache invalidation penalties are negligible compared to the long duration of the IOs.

7 Future work

We intend to examine [48] on the ARM platform. This method offers performance benefit on Intel architecture, and we intend to examine it on ARM architecture as well.

We expect further work to be undertaken in the ARM microvisor area.
We intend to utilize the microvisor in other ways. The hyplet presented in this paper is rapidly evolving in new directions. To name a few, we present the hyplet as a means to run userspace interrupts without overhead in Linux, and as an extremely fast remote procedure call (RPC). C-FLAT [44] - a control attestation system for embedded systems, was developed for TrustZone in devices with a minimal operating system. We will present an innovative technique to run C-FLAT in Linux with our new RPC. Kiperberg et al [45] presents a hypervisor-assisted atomic memory acquisition for the x86 architecture, we intend to present a port for hypervisor memory acquisition tool in ARM through the use of a microvisor.

The offline scheduler [40] is a technique to execute programs in an unplugged processor in Linux. We intend to demonstrate an evolution of the offline scheduler in the form of the offline microvisor.

8 Summary

This paper is a proof of concept that reverse engineering protection in ARM is applicable for CPU intensive workloads with minimal overhead. We achieved that by minimizing the number of context switches between the hypervisor and EL0. We can also assume that the encrypted sections are significant, and as such, the padding and the decrypting takes longer as the function size increases. For this, we recommend to minimize system calls and pre-map any memory that is accessed in the HYP context. For I/O intensive programs, we showed that the encryption penalty is relatively small compared to the I/O penalty, so our technology is most suitable for programs with high I/O rate.
Our solution proved stable during our internal testing. However, we also note that our ARM-based technology has not passed the same level of stability testing and penetration testing that the Intel solution has.

9 Compliance with Ethical Standards

Raz Ben Yehuda and Nezer Jacob Zaidenberg both declare that they own stock in TrulyProtect.
Ethical approval: This article does not contain any studies with human participants or animals performed by any of the authors.

References

1. Victor Costan and Srinivas Devadas, Intel sgx explained, IACR Cryptology ePrint Archive, 2016:86, 2016.

2. Balaji Balakrishnan, Matthew Hosburgh, and Patrick Neise, Securing the windows 10 GIAC enterprise endpoint.
3. Amir Averbuch Michael Kiperberg and Nezer Jacob Zaidenberg, Truly-protect: An efficient VM-based software protection. IEEE Systems Journal, 7(3):455–466, 2013
4. Niels Penneman Danielius Kudinskas Alasdair Rawsthorne Bjorn De Sutter and Koen De Bosschere, Formal virtualization requirements for the arm architecture, Journal of Systems Architecture, 144 - 154, 2013
5. Shaked Flur Kathryn E Gray Christopher Pulte Susmit Sarkar Ali Sezgin, Luc Maranget Will Deacon and Peter Sewell, Modelling the ARMv8 architecture, operationally: concurrency and ISA, In ACM SIGPLAN Notices, volume 51, pages 608–621. , 2016
6. HM Cantero S Peter and Segher Bushing, Console hacking 2010–ps3 epic fail, Chaos Communication Congress (December 2010), 2010.
7. Kevin Coogan Gen Lu and Saumya Debray, Deobfuscation of virtualization-obfuscated software: a semantics-based approach, In Proceedings of the 18th ACM conference on Computer and communications security, pages 275–284. ACM, 2011.
8. Rutkowska Joanna, Introducing blue pill, The official blog of the invisible things, volume 22, pages 23, 2006
9. Eldar Avigdor and Herbert Howard C and Goel Purushottam and Blumenthal Uri and Hines David and Smith Carey, Provisioning active management technology (AMT) in computer systems, Google Patents, US Patent 8 438 618, 2013
10. SMM loader and execution mechanism for component software for multiple architectures, Zimmer, Vincent J, Google Patents, 2005, US Patent 6848046
11. Winter Johannes, Proceedings of the 3rd ACM workshop on Scalable trusted computing, Trusted computing building blocks for embedded Linux-based ARM trustzone platforms, pages= 21–30, 2008, ACM
12. Winter Johannes, Trusted computing building blocks for embedded Linux-based ARM trustzone platforms, Proceedings of the 3rd ACM workshop on Scalable trusted computing, pages=21–30, 2008, ACM
13. Heiser Gernot and Leslie, Ben, The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors, Proceedings of the First ACM Asia-Pacific Workshop on Workshop on Systems, 19–24, 2010
14. Bernard Ngabonziza Daniel Martin Anna Bailey Haehyun Cho and Sarah Martin, Trustzone explained: Architectural features and use cases, Collaboration and Internet Computing (CIC), 2016 IEEE 2nd, International Conference on, pages 445–451. IEEE, 2016.
15. Amit Resh Michael Kiperberg Roee Leon and Nezer Zaidenberg, System for executing encrypted native programs, International Journal of Digital Content Technology and its Applications, 11, 2017.
16. Amit Resh Michael Kiperberg Roee Leon and Nezer J Zaidenberg, Preventing execution of unauthorized native-code software. International Journal of Digital Content Technology and its Applications, 11, 2017.
17. William Rosenblatt Stephen Mooney and William Trippe, Digital rights management: business and technology, John Wiley & Sons, Inc., 2001.
18. Gernot Heiser and Ben Leslie, The okl4 microvisor: Convergence point of microkernels and hypervisors, Proceedings of the first ACM Asia-Pacific workshop on Workshop on systems, pages 19–24. ACM, 2010
19. Thom, Stefan, Jeremiah Cox, David Linsley, Magnus Nystrom, Himanshu Raj, David Robinson, Stefan Saroiu, Rob Spiger, and Alastair Wolman. "Firmware-based trusted platform module for arm processor architectures and trustzone security extensions." U.S. Patent 8,375,221, issued February 12, 2013.
20. Nikolay Elenkov, Android security internals: An in-depth guide to Android's security architecture, No Starch Press, 2014.
21. Dan Rosenberg, QSEE trustzone kernel integer overflow vulnerability, Black Hat conference, page 26, 2014.
22. Prerna Mahajan and Abhishek Sachdeva, A study of encryption algorithms AES, DES and RSA for security, Global Journal of Computer Science and Technology, 2013
23. Amir Moradi, Mohammad T Manzuri Shalmani, and Mahmoud Salmasizadeh, A generalized method of differential fault attack against AES cryptosystem, International Workshop on Cryptographic Hardware and Embedded Systems, pages 91–100. Springer, 2006
24. Darrel Hankerson, Alfred J Menezes, and Scott Vanstone, Guide to elliptic curve cryptography, Springer Science & Business Media, 2006.
25. Asaf David and Nezer Zaidenberg, Maintaining streaming video DRM, In Proceedings of The International Conference on Cloud Security Management ICCSM-2014, page 36, 2014.
26. Marc Eisenstadt and Mike Brayshaw, The transparent prolog machine (TPM): an execution model and graphical debugger for logic programming, The Journal of Logic Programming, 5(4):277–342, 1988.
27. Rick Kennell and Leah H Jamieson, Establishing the genuinity of remote computer systems, In USENIX Security Symposium, pages 295–308, 2003
28. Michael Kiperberg and Nezer Zaidenberg, Efficient remote authentication, In Proceedings of the 12th European Conference on Information Warfare and Security: ECIW 2013, page 144. Academic Conferences Limited, 2013.

29. Kari Kostiainen N Asokan and Jan-Erik Ekberg, Practical property-based attestation on mobile devices, In International Conference on Trust and Trustworthy Computing, pages 78–92. Springer, 2011.
30. Karsten Sohr Tanveer Mustafa and Adrian Nowak, Software security aspects of java-based mobile phones, In Proceedings of the 2011 ACM Symposium on Applied Computing, pages 1494–1501. ACM, 2011.
31. Haryadi S Gunawi, Mingzhe Hao Tanakorn Leesatapornwongsa Tiratat Patana-anake Thanh Do Jeffry Adityatama Kurnia J Eliazar Agung Laksono Jeffrey F Lukman Vincentius Martin, et al, What bugs live in the cloud? a study of 3000+ issues in cloud systems. In Proceedings of the ACM Symposium on Cloud Computing, pages 1–14. ACM, 2014.
32. Amit Vasudevan Jonathan M McCune and James Newsome, Trustworthy execution on mobile devices, Springer, 2014
33. Jinmok Kim Donguk Kim Jinbum Park Jihoon Kim and Hyoungshick Kim, An efficient kernel introspection system using a secure timer on trustzone. Journal of the Korea Institute of Information Security and Cryptology, 25(4):863–872, 2015.
34. Min Zhu Bibo Tu Wei Wei and Dan Meng HA-VMSI, A lightweight virtual machine isolation approach with commodity hardware for ARM, In Proceedings of the 13th ACM SIGPLAN/SIGOPS International, Conference on Virtual Execution Environments, pages 242–256. ACM, 2017
35. Nelson Elhage, Virtualization under attack: Breaking out of KVM, DEF CON, 19, 2011.
36. Peter Szor. The art of computer virus research and defense, Pearson Education, 2005.
37. Christoffer Dall and Jason Nieh, KVM/ARM: the design and implementation of the Linux ARM hypervisor, ACM SIGARCH Computer Architecture News, 42(1):333–348, 2014.
38. Marcan Sven and Comex, Console hacking 2013–u fail it, In 30th Chaos Communication Congress (December 2013), 2013.
39. Roberto Mijat and Andy Nightingale, Virtualization is coming to a platform near you. ARM white paper, 20, 2011.
40. Ben-Yehuda and Wiseman(2013) The offline scheduler for embedded vehicular systems, International Journal of Vehicle Information and Communication Systems, Volume 3 pages 44–57
41. Maurice J Bach et al, The design of the UNIX operating system, volume 1, Prentice-Hall Englewood Cliffs, NJ, 1986.
42. Anatoli Kalysch, Johannes Götzfried, and Tilo Müller, Vmattack: Deobfuscating virtualization-based packed binaries. In Proceedings of the 12th International Conference on Availability, Reliability and Security, page 2. ACM, 2017
43. Jan-Erik Ekberg Kari Kostiainen and N Asokan, The untapped potential of trusted execution environments on mobile devices, IEEE Security & Privacy, 12(4):29–37, 2014.
44. Abera, Asokan, Davi, Ekberg, Nyman, Paverd, Sadeghi, and Tsudik. C-flat: control-flow attestation for embedded systems software, Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pages 743-754, 2016
45. Kiperberg, Leon, Resh, Algawi, and Zaidenberg, Hypervisor-assisted atomic memory acquisition in modern systems,
46. Sebastian Banescu Ciprian Lucaci Benjamin Krämer, and Alexander Pretschner, Vot4cs: A virtualization obfuscation tool for c. In Proceedings of the 2016 ACM Workshop on Software PROtection, pages 39–49. ACM, 2016.
47. Boaz Barak Oded Goldreich Rusell Impagliazzo Steven Rudich Amit Sahai Salil Vadhan and Ke Yang, On the (im) possibility of obfuscating programs, In Annual International Cryptology Conference, pages 1–18. Springer, 2001.
48. Kiperberg, Michael, Roee Leon, Amit Resh, Asaf Algawi, and Nezer J. Zaidenberg. "Hypervisor-based Protection of Code." IEEE Transactions on Information Forensics and Security (2019).