Hochschule für Technik, Wirtschaft und Kultur Leipzig

Fakultät Informatik, Mathematik und Naturwissenschaften

Master Computer Science

Masterthesis

to Earn the Grade

Master of Science (M.Sc.)

Evaluation of Intel Trusted Execution Technology for Use in a Partitioning Hypervisor

A Trusted Jailhouse

Submitted by: Benjamin Block

Matriculation Number: 57646

Munich February 12, 2015

Supervisors: Prof. Dr. rer. nat. Klaus Bastian

Dipl.-Ing. Jan Kiszka

Abstract

Modern hardware virtualization techniques on the x86 architecture make it possible to construct effective hypervisors with only a small necessary code base. Hypervisors like Jailhouse use this to provide safe segregation of the platform's hardware resources into distinct partitions. Using the guarantees given by the hardware and the small, verifiable code base of the hypervisor makes it possible to run tasks with mixed criticality together on the same hardware platform, put together with only commodity hardware.

Crucial for this function, and with that for the function of potentially safety relevant tasks, is that the hypervisor is started correctly, with the correct hypervisor-image, the correct configuration and on the actual hardware which provides the guarantees.

In this work, Intel's Trusted Execution Technologies, also shortened with TXT, are evaluated for the task to launch such a hypervisor during the runtime of a general purpose operating system. To prove this concept, a design to support TXT was created and subsequently implemented with the hypervisor Jailhouse as base. It is now possible to launch Jailhouse via TXT, and thereafter to verify its integrity with proven methods like remote attestation. Although this improves the confidence in the hypervisor setup considerably, it also became clear that TXT has a large amount of intrusive requirements, which it will force upon every software that wants to make use of it. It only becomes a valid option to use TXT in the envisioned use case, if these requirements are acceptable.

Eidesstattliche Versicherung

Ich erkläre hiermit, dass ich diese Masterarbeit selbstständig ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe. Alle den benutzten Quellen wörtlich oder sinngemäß entnommenen Stellen sind als solche einzeln kenntlich gemacht.

Diese Arbeit ist bislang keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht worden.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Benjamin Block, München, February 12, 2015

Contents

1 Gaining Trust in a Hypervisor

Virtualization solutions which execute multiple operating systems (or in general, bare metal applications) on top of and alongside with other software are today widely used in computer systems for a variety of tasks [WP10, Gol74]. They serve, for example, to support development and to test of new software — not least to develop and test operating systems [Bin06]. Virtualization is also used to raise efficiency in data centers by running more than one system on real hardware, and thus it lowers the time this hardware spends in idle [KNS13]. In doing so, those solutions also guarantee that the virtualized systems are isolated from each other, so that they can't access each other's information or worse, influence each other maliciously [BM13].

But until some years ago, it was not possible to develop such solutions in a way to allow them to execute the virtualized software (*virtual machines*, abbr. *VM*) unchanged and directly on processors of the x86 architecture. This was largely due to some operations that would, when executed natively on the processor, expose systems states without the ability to trap them, or they would fail silently without generating errors, and thus they would fail the isolation [NSL+06]. Softwares like *Xen* or *VMware* had to deploy other solutions to make the virtualization possible [AA06, BDF+03, BDR+12].

They also had to develop ways to make it possible for the VMs to use I/O devices. While some decided to fully emulate I/O, and thus also made it possible to multiplex existing devices, others had to spend this unwanted effort to uphold the isolation [AJM+06].

All this overhead meant that the performance of the virtual machines would suffer [ByMK+06]. This lead to the implementation of extensions for the x86 architecture by both Intel and AMD [NSL+06, Kle09, AMD05] — Intel named their extension *VT-x* and AMD theirs *SVM*. They allow the virtual machine monitors (the part of the virtualization solutions that creates and maintains the environment for the virtual machines, abbr. *VMM*, also called *Hypervisor*) to let the virtualized software run directly on the processor. Any operation that would change an important system state or leak inappropriate information can now be trapped and handled correctly by the VMM. Further introductions of additional memory management units between I/O devices and their connection to the main memory via DMA (*IOMMU*s) have made it possible for VMMs to let virtual machines use these devices directly as well [AJM+06].

To use these extensions, the VMM has to have sufficient privileges, it has to be executed with privilege level 0 (often also referred to as ring 0) [AMD05, Int14b]. Even more, once the system activates the extensions, the VMM can overrule any other piece of software — including the operating system that may have been executed before the activation [Sin14a]. This means that every software which is executed thereafter on top of it — possibly multiple complete operating systems, together with their own software stack — has to rely on the proper function of the VMM.

This provides the motivation for this work. The first step in ensuring that the VMM — or any other piece of software — executes properly, is to make sure, before executing it, that the loaded code with all its inputs is the expected. The same problem can already be seen in the execution of operating systems and a software stack on top of them — correct execution relies on the proper function of the OS. Along with this task comes the next problem: how can a third party be convinced — may it be the user of the system or any other computer — that the correct software was started and is still running? It can't trust any results or commands from this software or, in case of a VMM, any of the VMMs without this knowledge.

This is exactly the problem that the Trusted Execution Technology from Intel (abbr. *Intel TXT*) tries to solve [Gre13, Int14c]. TXT was introduced by Intel in 2007 and adds new instructions to x86 to support the *trusted execution* of arbitrary software, at any point during the lifetime of a system — not limited to either VMMs or OSs. To support its task, it makes heavy use of the Trusted Platform Module (abbr. *TPM* [CYC+08, Cor10]) to securely save information and to provide the ability to attest that the expected software was executed.

Currently TXT is only used in few software systems, most of them use it in the context of *secure boot* (as *replacement* for the same functionality provided by UEFI). They activate it as part of the boot process to record and test the code of the OS or VMM that is about to be started on the bare metal [FG13].

This work will evaluate it as means to start a VMM during the lifetime of an already running OS. In the presented scenario, the OS needs to be kept unimpaired, and will continue to execute as one of the started VMM's virtual machines. A design and implementation, confirming to these requirements, has been created with the hypervisor *Jailhouse* [Jaib, Kis14] and the operating system *Linux*. This work will present both the design and implementation, and a throughout analysis of what Jailhouse can guarantee by using Intel TXT to start. Furthermore, it will show how a user can gain trust in the system once it is started, and what impact the additional work for using TXT has on the hypervisor.

1.1 Structure of this Work

The following chapters of this thesis are organized in the following way. Chapter 2 on page 5 will give a more in-depth introduction into the topic virtualization; it will define the problems VMMs face and how recent hardware extensions on x86 improved that situation; finally, it will present the VMM used as example in this work: Jailhouse. Chapter 3 on page 25 will then present the motivation as to why Intel TXT is considered to be used to start Jailhouse, and how exactly TXT works as hardware extension. After establishing these fundamentals, Chapter 4 on page 53 will elaborate on what others have done in this area. Thereafter, Chapter 5 on page 61 will present the design developed to solve the task of this work. Chapter 6 on page 75 will then present the implementation created according to this design, together with measurements about its size and performance. In Chapter 7 on page 101, the design and implementation will be evaluated for their security: against what malicious attacks can they defend themselves by using TXT in the way proposed. The final two chapters, 8 on page 117 and 9 on page 119, will discuss what future work in this field is still necessary and conclude the results that could be made during this work.

2 Virtualization of Hardware

This chapter will present the necessary fundamentals to understand how the virtualization solution Jailhouse works. Jailhouse will be used as the main example in this work, and thus also sets a large share of the requirements for the design and implementation created with it. Apart from this, parts of the shown techniques will later also be used as means in the security analysis to accomplish certain properties.

The first Section 2.1 will define what this work understands under the broad term of virtualization and what requirements this poses for a solution aiming to implement this concept. How these requirements can be fulfilled with recent architecture extensions to x86 will be presented in Section 2.2 on page 7. Finally, Section 2.3 on page 15 will give an in-depth overview of the main hypervisor example in this work: Jailhouse. It will show how Jailhouse is different from other solutions, how it handles the virtualization of the system and with that, what requirements it imposes on a design and implementation of Intel TXT for it.

2.1 The Use of Virtual Machine Monitors

2.1.1 Definitions

Virtualization in computing is a very broad term, used for a variety of topics. This work will use the following as the general definition for the term [Sin04]:

Definition 2.1. ***Virtualization*** is a framework or methodology of dividing the resources of a computer into multiple execution environments, by applying one or more concepts or technologies such as hardware and software partitioning, time-sharing, partial or complete machine simulation, emulation, quality of service, or others.

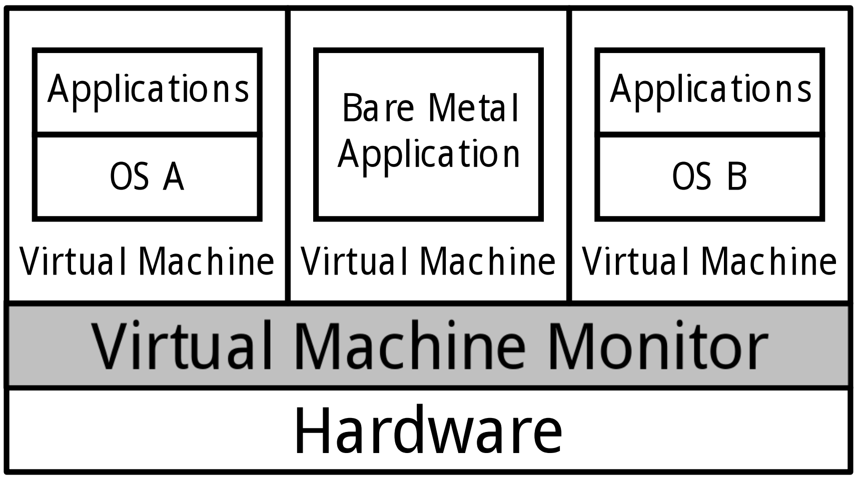

FIGURE 2.1: Organization of virtual machines on top of a VMM virtualizing the complete hardware of computer system.

This methodology is frequently applied to different resources of a single computer, such as memory, the processor or I/O devices; most commonly by the operating system running directly on the hardware. This makes it possible to run more than one application on top of it, without the need for these applications to know of each other or the limitations of the actual hardware. This requires that the operating system has full access to the actual hardware [TB14].

What this work uses the term virtualization for, is to create a virtual version of a whole computer, so that it becomes possible to run multiple operating systems — or more generally, applications that assume direct hardware control (*bare metal* applications) — next to each other, each on their own virtual version of the system. One such virtual instance of the system is commonly called a *virtual machine* (abbr. VM) and was formally specified in 1974 by Popek and Goldberg [PG74].

The software used to manage the different virtual machines on one computer is called the *virtual machine monitor* or *hypervisor*; an example of how such a system can be organized is shown in Figure 2.1. Popek and Goldberg defined a VMM as follows:

Definition 2.2. A software system is considered a ***virtual machine monitor*** if it has the following characteristics:

1. *Fidelity*: The VMM provides software running on it an environment essentially identical with the original machine.
2. *Performance*: The software running in this environment shows at worst only minor decreases in speed.

3. *Control*: The VMM is in complete control of all the hardware resources of the underlaying computer system.

By requiring an "essentially identical" environment, it is left open that there may be differences in timing and system resources available to the virtual machine. Both stem from the fact that the original system is shared between multiple virtual machines and thus can't provide the same resources as if it was used directly. For example, there will be less memory and processor time available for a single VM.

To achieve the second characteristic about performance, it is necessary to run most of the instructions of the VM directly on the real processor. A VMM that would uses an interpreter to execute the instructions may have to use hundreds of physical instructions for the interpretation of just one instruction of its VMs [AA06]. This rules out the use of an emulator.

The final characteristic makes sure that VMs can only ever use resources explicitly allocated to them — even more, the VMM has to have the ability to recover any resource from a VM, if that is necessary. This also means, any access to another VM's resources which are not explicitly shared has to be suppressed by the VMM, and thus makes it possible to isolate them from each other.

Implementation approach The classic implementation approach for a VMM to fulfill those characteristics is *Trap and Emulate*.

A more comprehensive description of this approach, and why until recently it was **not** possible on x86 to implement it directly, can be read in Appendix A on page 121. VMMs did exist, but they had to use more complex and slower algorithms [VMw09, BDR+12]. This situation only just changed with the introduction of the extensions described in the following section.

2.2 Hardware Support for Virtualization on x86

To make it possible on x86 to implement efficient VMMs, with only a small amount of source code (compared to the solutions made before), both Intel [NSL+06, Int14b] and AMD [AMD05] released extensions for their x86 architectures to support the original trap-and-emulate approach. This section gives a short overview of how Intel solved the problem with their extension *VT-x* (AMD works conceptional in the same way, but is differently implemented and thus incompatible).

2.2.1 CPU-Virtualization

Essentially, *VT-x* introduces two new forms of operations to the x86 architecture: *VMX root operations* and *VMX non-root operations* (*VMX* stands for *Virtual Machine eXtensions*). VMX root operations are very much like operations without the extension. The only differences are, once activated, root operations have access to a new set of instructions (the VMX instructions) and certain control registers become become limited (some bits in `CR0` and `CR4` become fixed). These operations are intended to be used by the VMM [Int14b].

The VMX non-root operations are intended to be used by the VMs, or *guests* as they are also commonly called. They also behave much like the normal x86 instructions. It is, for example, possible to operate under all four privilege levels with them. But they add the ability for the VMM to trap all sensitive instructions executed by the guest — such a trap is called *VM exit*.

Trap-and-Emulate using VT-x To enter these new operation modes, the VMM has to execute the instruction *VMXON*. It then executes with the new VMX root operations and has access to the new instruction to setup and configure its guests. Subsequently, it can decide to enter a guest by doing a *VM entry*. This will set the processor into the VMX non-root operation mode and then continue the execution in the guest as long as no *exit reason* is reached — a trap for example. Figure 2.2 on the next page illustrates this life cycle based on a VMM with two different guests.

The transition between root- and non-root is controlled by a structure called the *Virtual-Machine Control Structure* (abbr. *VMCS*; this acts as a shadow structures in the trap-and-emulate scheme). It includes an area controlling the state of the guest upon a VM entry (the *guest-state area*), an area specifying the state that is restored when a VM exit occurs (the *host-state area*) and finally some control fields to configure when and why a VM exit shall happen (the *VM-Execution Control Fields*).

Figure 2.2 on the facing page also shows how this structure is used during the life cycle of a VMM. After the VMM has executed `VMXON` and entered the root operations, it will configure the VMCS by using the new instructions that have been added for this (`VMCLEAR`, `VMPTRLD`, `VMREAD` and `VMWRITE`). It writes its own processor state into the host area, and the state of the guest into the guest area. This includes all crucial operation state information of the processor, such as:

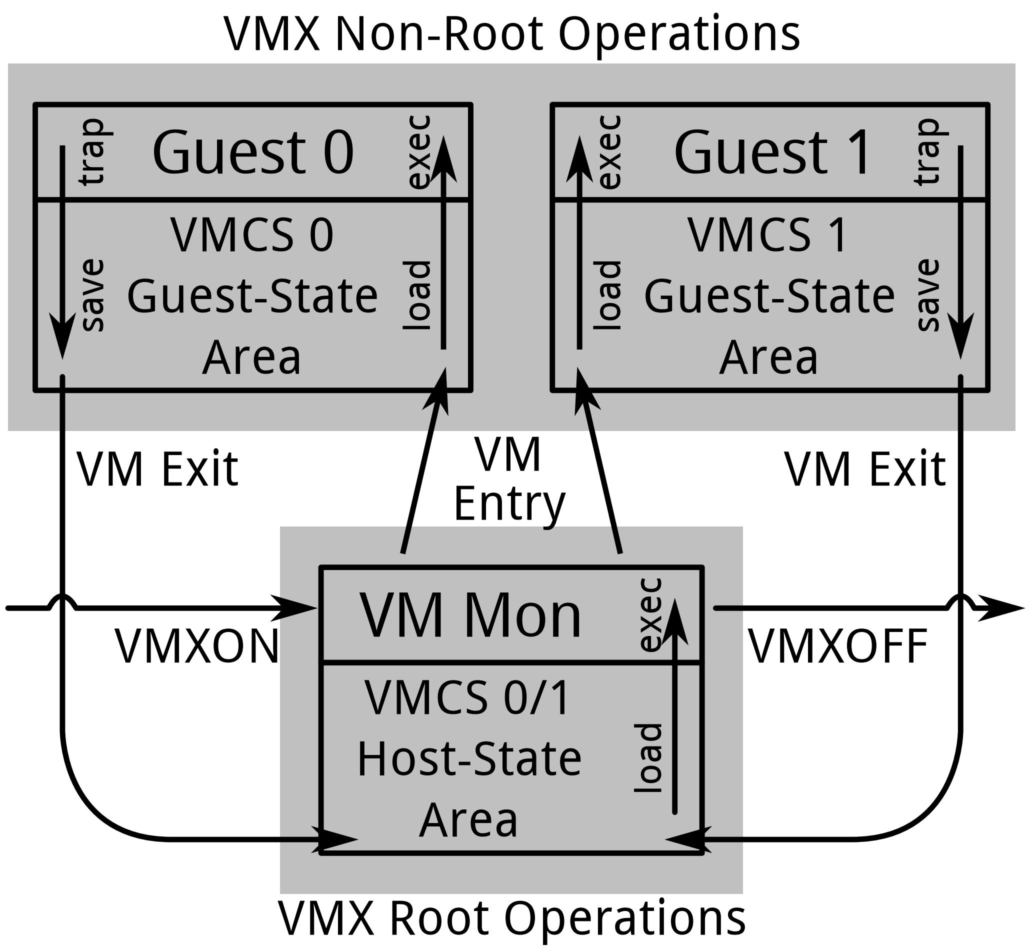

Figure 2.2: Diagram showing the life cycle of a virtual machine monitor using the VT-x extension. Each combination of VMM and guest has its own VMCS, with its own set execution elds; the host-state area has to be shared manually between them.

- control registers,
- segmentation registers,
- some model specific registers (*MSR*s; for example the MSR *EFER*),
- important execution registers, like the instruction pointer,
- and the descriptor registers.

Next to these states, the VMM also specifies what later shall cause a VM exit. Again, this list is made up from several components, the most important include:

- execution of several different sensitive instructions,
- interrupts during the guest-execution,
- accesses of sensitive processor registers,

- I/O via the port I/O mechanism and others.

Most of these *exit-reasons* can also be specified more precisely than only on the base of the whole class of operations (which would be the case in the classic trap-and-emulate scheme). It is, for example, possible to enumerate the exact I/O ports a guest can access uninterrupted via port I/O, and which others will cause a VM exit. This way it is possible to allocate resources to a VM without the need to intervene later — saving VM exits, which are quite costly [FO06].

When this is done, the VMM can activate the configured guest via the instruction `VMLAUNCH`. This instruction won't return on success, it will instead load the specified states from the guest state area and the processor will continue executing in the non-root operation mode, until one of the exit-reasons is met.

Lastly, once such an exit-reason is met, the processor will proceed with a VM exit and reload the states saved in the host area of the VMCS and return to the VMM. It can then examine the exact reason and emulate the effect appropriately, like in the classic trap-and-emulate scheme. This process is continued for the whole life-cycle of the VMM and ultimately turned off by a call of `VMXOFF`.

Memory Tracing using Extended Page-Tables Although this would already meet Popek and Goldberg's theorem (A.1 on page 122) [NSL+06] and remove the need for implementing a complex binary transformation engine inside the VMM, it was found that it would not add much performance advantage to a VMM using it [AA06]. The main reason for this was the need to still implement a page level memory tracing (described in A on page 122). This meant that every access to a critical memory area in a guest (e.g.: page tables or MMIO areas) had to be trapped and would cause a VM exit; this added a considerably large overhead to the execution [VMw09].

This shortcoming was again addressed by both Intel and AMD [AMD08]. Intel's implementation is called *Extended Page Tables* (abbr. *EPT*). The more general term, which addresses both implementations, is *Second Level Address Translation* (abbr. *SLAT*). It extends the paging algorithm used during VMX non-root operations with an additional address translation step.

Usually, when paging is activated, addresses are translated two times: the addresses used in the software are *virtual addresses* and are translated into *linear addresses* using the segmentation mechanism of the protected mode (although today, virtual addresses are usually resolved into the same linear addresses to remove complexity and because segmentation is only partially supported by x86_64); these are then

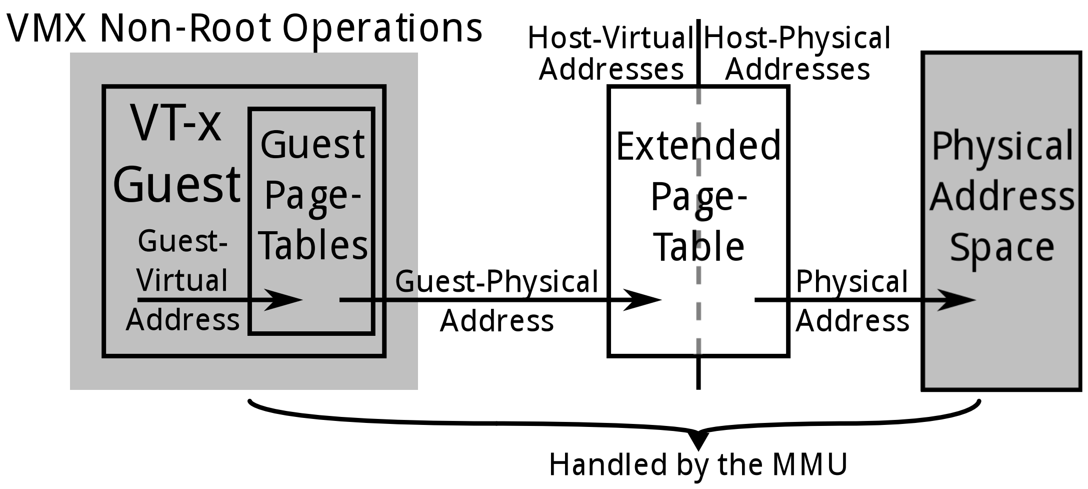

FIGURE 2.3: Diagram showing the resolution of a virtual address of a VT-x guest into a physical address of the host using the extended page tables.

translated again using the activated page-table and thus resolve into the final *physical address* [Int14b].

SLAT adds the notion of *guest-physical addresses* to this algorithm. During non-root operations, addresses are translated using the same procedure as described above, but the result is not an actual *physical address*, but instead a *guest-physical addresse*. This address is then translated again, using a separate page-table configured by the VMM (the *EPT*), and only then resolves into the actual *physical address*.

This process is depictured in Figure 2.3. Both translations, the one from guest-virtual into guest-physical addresses and then the one into host-physical addresses, are handled by the hardware memory management unit (abbr. *MMU*). The EPT itself is assembled during the configuration of the guest or later on demand by the VMM. It can not be accessed by the guest during the non-root operations — it is completely transparent to the guest; it just programs the paging as it would running directly on the hardware. Because the VMM can exactly choose which physical resources it wants to allocate for the guest, it has no longer any need to intervene later on when the guest is executed, and thus it can completely remove the memory tracing and traps for accesses to the page-configuration registers.

Further extensions to VT-x (the *Unrestricted Guest Mode*) made it even possible for guests to run completely without paging, in the *real-* or *unpaged protected- mode*. During those, the EPT is still enabled and the MMU still translates every memory reference done by the guest, but the guest itself does not have to employ any translation scheme. Hence, the control characteristic is still satisfied, but the VMM can deactivate even more traps and thus has to handle even fewer VM exits.

2.2.2 Virtualizing Device I/O

Another difficult problem for VMMs on x86 is to maintain the control characteristics for device I/O [WRC08, WSC$^+$07, AJM$^+$06]. By using VT-x in combination with an EPT, it is possible to safely allocate I/O ports and MMIO regions directly to a single VM. I/O ports are handled as described in 2.2.1 on page 8 and MMIO regions by enabling the SLAT — a VM may only access a MMIO region that is mapped into its guest-physical space.

But devices usually also deploy some form of DMA to save processor time, they read and write data directly into the main memory. These accesses are not managed by the processor's MMU, and therefore they are also not affected in any way by the SLAT or any other paging deployed by the processor. Guests could abuse this by programming the assigned devices with wrong physical addresses — violating the control characteristic.

Emulation and Paravirtualization Two of the most popular approaches to solve this problem are to completely emulate the devices or to provide a special ABI with logic device function — this this called *Paravirtualization* [AA06, Rus08, Jon10]. An overview for both concepts can be seen in Figure 2.4 on the next page.

Both approaches share that the real device is not exposed in any way to the guests of the VMM, and thus they solve the problem with DMA by not letting the guests access the devices. Instead, the VMM has its own complete device driver and exposes chosen functions of it via a added interface to its guests — how many functions and how they are accessed is based on the chosen method.

Direct I/O Access via an IOMMU The downside of both of these approaches is that they require the VMM to implement its own drivers and a device abstraction layer. While this is desirable for VMMs that want to present guests with more resources than actual available or share one resource between multiple guests, it adds unnecessary complexity for others. But to be able to give the guests direct access to the devices, there has to be a mechanism to prevent them from programming the devices with incorrect addresses.

This is the motivation for another extension recently added to x86: *IO Memory Management Units* (abbr. *IOMMU*). The concept behind those is parallel to the

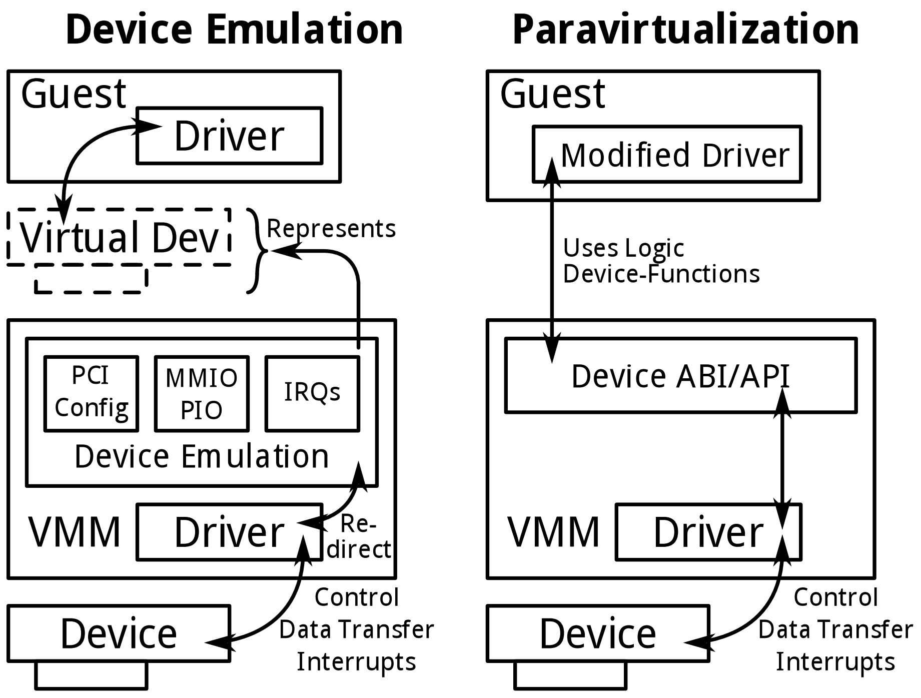

FIGURE 2.4: Diagram showing the architecture of a VMM providing virtualized devices via a software implementation. The left VMM uses a software emulation to provide its guest with an identical interface so it can use an unmodi ed driver. The right VMM provides its guest with a de ned device ABI and API that provides logic functions like *send network package*; the guest has to use a modi ed driver to use the functions provided by the API with the ABI.

one behind SLAT: an IOMMU adds an address translation step to memory accesses of I/O devices.

Rather than a guest-physical address, the IOMMU introduces the notion of a *device-physical address*. Those are the addresses the device gets programmed with, and thus whenever the device makes an access via DMA, it will use these addresses, but it won't access the physical memory directly. Once activated, the memory access will be routed through the IOMMU, which is locate between I/O devices and memory — in case of Intel's implementation *VT-d*, as part of the north bridge [Int13, AJM^{+}06].

During the access, the IOMMU will identify the device (for example by its PCI *(Bus, Device, Function)*-tuple) and select the programmed page table. Each device can have its own page table, or some can share a common one. This table is then used to translate the device-physical address into a *host-physical address*, which is then used to access the memory.

This setup is, again, programmed by the VMM and transparent to the devices, just as the EPT is to the VMM's guests. The VMM could, for example, decide to map

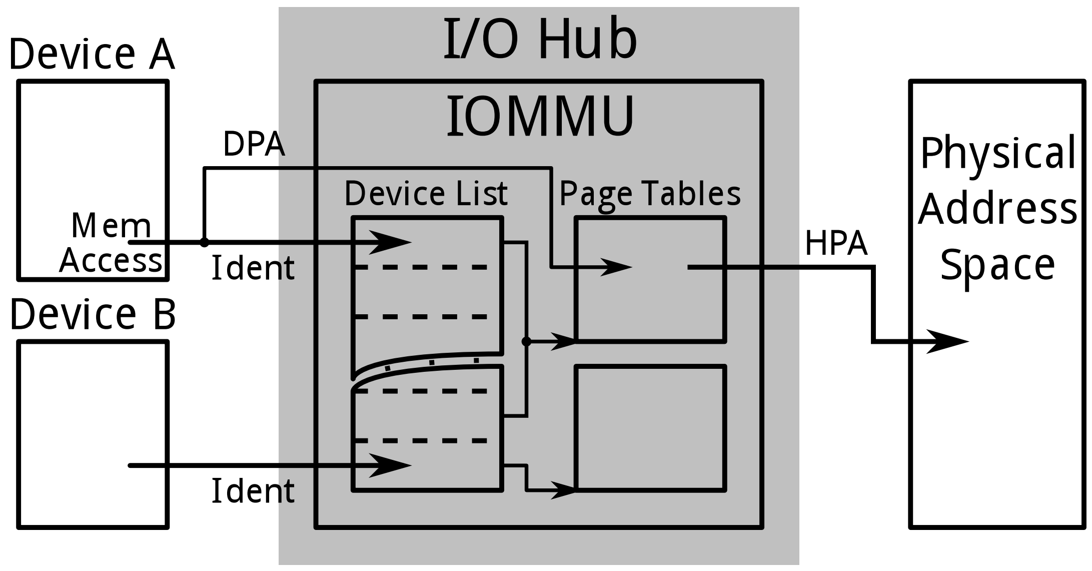

FIGURE 2.5: Concept of an I/O device making a direct memory access via an immediate IOMMU. *DPA* stands here for *Device-Physical Address* and *HPA* for *Host-Physical Address*. The *Ident* is given implicitly and may for example be the PCI *BDF*.

the exact same memory region in the device's page table, with the same translations, as it does in the EPT. This way, the guest can use its own guest-physical addresses to program the devices, and accesses will translate into the same memory locations for both, and thus relieves the VMM from intervening in this process.

Other than that, *VT-d* can also be used to handle the device's interrupts (*Interrupt Remapping*). Again, this is done by applying another translation table. This table is used to redirect interrupts generated by a device to a specific processor, and with that to the guest running on it [Int13]. With this, it prevents inappropriate interrupts to unrelated guests or information leaks.

With both features together and *VT-x*'s ability to moderate accesses to in-memory config areas, it is possible to assign devices directly to a VMM and moderate all the influences guests can have on other parts of the system via DMA or interrupts. It won't however solve all problems that can arise from direct assignment. It is possible that devices employ other communication schemes than DMA, or that a guest brings a device into an unrecoverable fail-state; in both cases it could still violate the control characteristics of the VMM. If such a behaviour is possible for a device, it has to be handled specially[1].

[1]It was not possible to nd a general purpose solution for such a special device behaviour, it is doubtful there is any.

2.3 The Jailhouse Hypervisor

Most of the virtualization solutions today make at least use of some of the presented extensions [WP10]. The solutions most commonly known use them to provide a degree of *over-provisioning* to their users [KNS13]. They aim to run multiple, full operating systems as guests and supply each of these guests with a full range of usable processors and I/O devices. This way, each guest can have its own network adapter, its own set of processors and its own memory. For a VMM running multiple such systems on one physical host, it is hard to provide each of those resources dedicated to only one virtual machine, and thus it is not possible to allow uninterrupted direct access.

To solve this problem, they apply a similar set of algorithms as a normal operating system. Processor time is shared via a scheduler, memory is virtualized via EPTs — interpreting a VM as a normal application running on an operating system, the EPT can be seen as normal page table — and for I/O there is often a mix of emulation and paravirtualization used. Combined, a VMM can allocate a single resource of the host to multiple of its VMs or even over-allocate the resources of the host. If it satisfies the characteristics defined in 2.2 on page 6, the software running on the VMM will not notice anything and won't be able to have influence on others — ignoring impacts in efficiency. While this can increase the efficiency of one host and lower the required physical hosts [KNS13], it also means that it becomes harder to predict what time- and resource-guarantees a VMM can make for its guests [CAA08, CGFC10, XLG$^+$13].

Jailhouse aims for a different goal, it aims to provide its guests with a strong isolation and guaranteed time constraints. Both with only a small and potentially verifiable code base. To make this possible, it doesn't employ *over-provisioning*, any form of scheduling, and resource-sharing only if explicitly configured [Kis14, Sin14a, Sin14b, jaia].

Instead, it splits the available hardware into distinct (static) *partitions*, and once one of these partitions is allocated to a guest, it is not changed anymore. This approach has already been used in the past on other architectures than x86, for example, IBM's *System z* provides a similar feature called *Logical Partition* (abbr. *LPAR*) [BCDB$^+$14].

To achieve the small code base, Jailhouse refrains from using software to intervene as much as possible. Rather then that, it uses the presented hardware features on x86

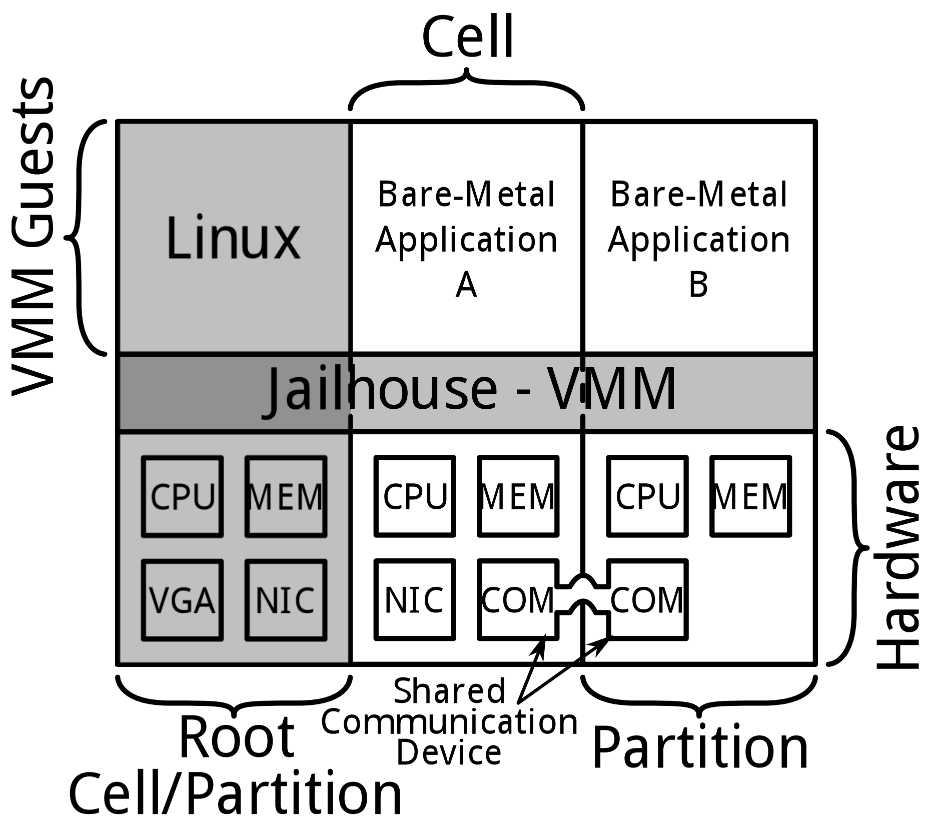

FIGURE 2.6: Overview of Jailhouse's base architecture.

to achieve the isolation, and only traps instructions when absolutely necessary.

Architecture of Jailhouse The basic architecture and how Jailhouse organizes a system can be seen in Figure 2.6.

As shown, the available hardware of the system is completely split into separate *partitions*. One such partition must at least have a single processor and some memory assigned (otherwise, it would not be possible to execute a VM on top of this partition). Other than that, it can be made up from any other available processors, memory and I/O devices. But once a resource is allocated to a specific partition — other than the root partition, which will be explained below —, it is not possible to change or interchange it with a resource from another partition. Otherwise, the user is free to assemble the partitions as he sees fit. He even could explicitly *share* memory and I/O devices between partitions — although this may be difficult to handle for the guests running on them. The partitions are also frequently called *cells* in the context of Jailhouse — they are essentially the same as a VM with a static set of resources.

During the start of Jailhouse, the main hypervisor code will be loaded by a Linux kernel module, alongside with a binary configuration containing a detailed hardware description of the target system — it lists processors, memory and devices, all with their respective access rights. It will then give control to Jailhouse, which will use the given configuration to bootstrap itself on the target machine (it won't use information provided by Linux).

In this bootstrap phase, the VMM will create the *root cell*. This cell will later represent the Linux which started this process. The VMM will assign all CPUs, the memory and all devices to this cell which are described in the given configuration (nothing more, left out resources will not be available). This cell will then finally be programmed with the state of the previously running Linux as guest state, and the VMM will continue running it as guest on top of it (more details of this process will follow later in this section).

Whenever an additional partition is created for another guest, the assigned resources are gradually taken away from the root cell (it is also possible to assign resources to a partition that were not previously assigned to the root cell). The resources will only be given back to it, in case those new partition are destroyed again (if originally configured this way). For most devices — with hotplug functionality — this can be done without modifications to the Linux running in the root cell (there are some corner cases; for example memory is not as easy to handle, the solution for this will be given later in this section)

Virtualization-Techniques used in Jailhouse To implement the VMM-functions, Jailhouse makes heavy use of the x86-extensions introduced in this chapter (or equivalent functions on other architectures like ARM). It requires at least a processor with support for VT-x with the EPT-feature and an IOMMU, or the equivalents on AMD's processors.

The guests running in the cells of Jailhouse — this includes the Linux running in the root cell — are executed with the techniques described in 2.2 on page 7. Using the IOMMU, they also get direct access to the allocated I/O devices in their partition. In order to lower the complexity of the hypervisor, Jailhouse tries to not emulate any devices. Exceptions to this rule are parts of the interrupt controller, device configuration spaces, a virtual inter-cell communication-device and interrupt remapping for the root cell.

During the runtime of a non-root guest, Jailhouse tries to do as few VM exits as possible.

The mappings in both the EPT and the IOMMU page table are never change during the runtime of a guest. This means, there will never be page-faults that would require exits and handling (other than page-violations).

Processors are handled in the same way, they are statically assigned to certain guest cells — making scheduling of the them needless. In combination with the interrupt remapping tables of the IOMMU, this makes it also possible to statically assign device-interrupts to certain partitions, which makes exits because of them unnecessary as well.

In general, interrupts only ever cause VM exits, if they could compromise the isolation between the cells (for example, in case of inter-processor interrupts, Jailhouse has to verify that the target processor is in the same cell as the source).

This all makes it possible, unless a guest deliberately accesses sensitive control information, to execute non-Linux guests with zero VM exits.

Other than not emulating any devices, it is also important to note that Jailhouse does not support the *start* of unmodified guests on x86 (Linux can *run* unmodified, if it runs in the root cell). This is because Jailhouse does not emulate a BIOS firmware and has a non-standard reset vector (on x86 this is normally located at `0xFFFFFFF0`, Jailhouse places it at `0x000FFFF0`)

Usecases for Jailhouse It is not the intention of Jailhouse to run multiple full operation systems on one system; this would also be hard to achieve because of the limited resources — multiple full OSs would require a lot of redundant hardware in a machine. The main usecase for Jailhouse are systems that run applications with diverging *criticality* which need to be separated from each other in a safe way.

For example, a system that controls an engine and displays information about it to an operator could be split into two separate cells, one for each task. The engine control, a safety critical task, could run in its own cell and use a dedicated interface card to communicate with the engine (this card would be assigned to the control cell exclusively). The root cell could continue to run Linux and use the existing graphical infrastructure to display the various engine value. It would receive them from the engine control through a defined communication channel. Jailhouse separates both cells in a safe way through the use of the hardware extensions and can thus guarantee that the root cell may never have influence over the engine or the interface card of the control cell. Even if Linux would crash for a reason, this would only let the uncritical task of displaying the information fail. The engine control could continue to run and safely shut down the engine to react upon the failure of the other cell.

An other field where Jailhouse can be very helpful, is to consolidate legacy single-core operation systems into one multi-core system (e.g.: different RTOSs — no general purpose OSs). Each core of the system, along with a small amount of memory and necessary devices, could execute its own OS and with that continue its lifetime, without the need to re-implement (and possibly re-verify) those tasks in new environments. The Linux in the root cell could then be used as interface to collect and aggregate information of the other OSs, but thanks to Jailhouse it would be unable to influence them inappropriately.

2.3.1 Running Jailhouse on Linux

To better understand the later presented solution, to execute Jailhouse in a trusted way, this section will briefly show the key steps of starting Jailhouse in the normal way.

Jailhouse's Components An execution of Jailhouse makes use of 4 major software components: the Linux kernel driver, a management utility, the hypervisor image and the hypervisor configuration. An overview of how they interact can be seen in Figure 2.7 on the next page.

Because Jailhouse makes use of VT-x and other hardware extensions, it needs to be executed with privilege level 0 [Int14b] — the highest. Under Linux, this level is restricted to the Kernel and thus the need for a ***driver*** that runs in the context of the Kernel [Lov10]. Despite that, it is important to note that Jailhouse itself is not part of the Linux Kernel — unlike, for examples, the kernel's own hypervisor KVM [KKL+07]. The kernel is only used to start the hypervisor — to gain the required privilege level, reserve the necessary memory and gain control of all the processors — and to interact with it, once it is started.

The implementation of the driver is also not part of the kernel itself, but also part of Jailhouse and build alongside with it — not as part of, but individual component. The resulting module can be loaded at any time before the start of Jailhouse and will then present the *root* user with a device node as interface (this node is programmed with the usual system calls).

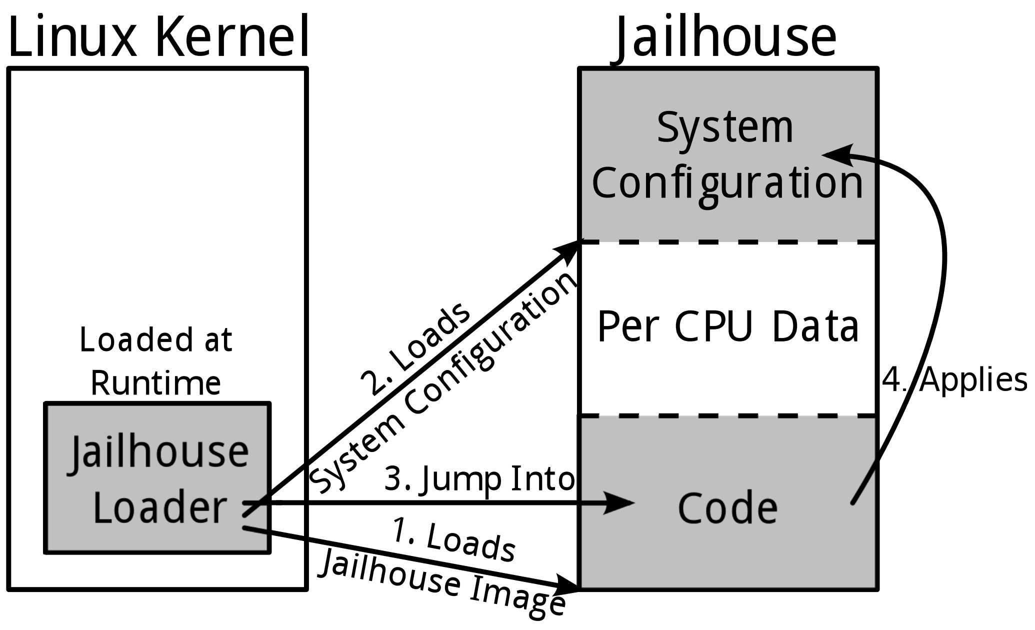

FIGURE 2.7: Diagram showing Jailhouse's software components and how they interact with each other during the startup of the hypervisor.

For the interaction with the driver, a small set of ***utilities*** is provided along with it. They make use of the provided device-node and are the main user-interface at this point.

The ***hypervisor*** itself — represented as "Code" in the figure — is represented as a simple binary image. This image is platform specific and only contains the components required for the target system — Intel, AMD, ARM. Which exactly, is decided at compile- and link-time. It does not though contain information about the runtime itself — as for example code location, processor count or available I/O devices.

Those information are stored in the ***configuration***. This file will contain a hardware description of the system Jailhouse is executed on (this may contain fewer resources than available; an overview is shown in Table 2.8 on the facing page). Instead of using extensive device- and system- probing via *ACPI*, PCI configuration areas, BIOS or other such methods on x86, Jailhouse relies mostly on its config and applies detection only where it is necessary — for example, to detect runtime information that are to inconstant for long-time storage. As of the moment of this writing, the only exceptions to this, that are taken from Linux, are the logical processor IDs the Linux kernel assigns at its boot-time. These are given as arguments, from the driver to the hypervisor, and are reused as identifier in the hypervisor.

Configuration Part	Description
Hypervisor Location	The (physical) location and size of the hypervisor in the memory during runtime.
Processors	A bitmap enumerating all available processors; the position corresponds to their logical ID.
Memory Regions	An exact listing of all memory regions, containing usable RAM and regions reserved for ACPI, PCI and other system components. Each defined region also contains the expected usage permissions (read, write, exec, DMA) and usage type (I/O, comm. region, RAM).
IRQ Chips	A listing of all available IRQ chips and their MMIO location in the memory (e.g. IOAPICs).
PIO Bitmap	A description of the whole PIO space with a bit for each port, describing if accesses to this port are allowed or not.
PCI Devices	All available PCI devices, their identifier, type, PCI capabilities and the IOMMU they are behind.
Platform Details	More details that are specific to the used computer platform, like the amount of available interrupts or number of available IOMMUs.

Figure 2.8: Table showing the parts and their description of a Jailhouse con guration le.

Loading the Hypervisor Starting Jailhouse, once a complete configuration is created, is straight forward (see figure 2.7 on the preceding page).

Once the driver module is loaded into the Linux Kernel, the root user can *enable* Jailhouse by sending an IOCTL system call to the created device node, with the chosen configuration as argument. The driver will then map the configured physical hypervisor location into the kernel's page table and load the remaining components into that area (the virtual address space of Jailhouse always start at `0xfffffffff0000000`). This location is not allocated in Linux, but is reserved for Jailhouse via a boot-time argument of the Linux kernel, and is thus also always at the same location for a given configuration[2].

The hypervisor image (the code) is loaded at the beginning of the memory area. Following the code, the driver allocates a small amount of memory for each available processor (labeled "per cpu data" in the diagram). This space — seen as simple array — is later used to store information unique to a specific processor. Finally,

[2]This may be used at a later point during the development to check the integrity of Jailhouse in safety critical applications, by calculate stable checksum over this area.

the driver loads the given configuration completely and unchanged into the ensuing space.

At this point, all component are loaded into the memory and Jailhouse can be started.

Running Jailhouse To do this, the driver has to make a function call on *every* available processor into the address space of the loaded code. This function represents the *entry*-point of the hypervisor and its address is specified at the beginning of the loaded image (in a header). At this point, the hypervisor will only return full control over the hardware back to Linux, in case an error happens during the startup or at the time the hypervisor is shutdown again — the driver relinquishes control over the system on behalf of the Linux kernel.

The entry function will first store all registers which represent the current operation mode and computation state of the kernel, into the area of the particular processor. With those saved states, the VMM will later be able to continue running the Linux as a guest, without any notable discontinuities.

After that, the entry path will continue by checking and promoting Jailhouse's own desired CPU and hardware state, including the change from Linux's page table to a newly created one. This will eventually lead to the setup of the VT-x and IOMMU structures. For this last step, it will put the previously stored processor states into the guest-state area of the VMCS, and its own into the host-state area; and thus, when the VM enter is done, it will continue running Linux as guest with the exact state it had when it entered the entry function.

The lengthly process that happens in between those calls is out of scope of this work; a far more comprehensive description of it can be found in [Sin14a, Sin14b]. Here, we will only note two more points.

First, at the point where the entry function is called, the processors still use the page table of the Linux kernel. Jailhouse will exchange this table for its own and also deploys its own simple page allocator to manage its virtual address space and later on, that of its guests.

Secondly, the EPTs and IOMMU page tables will only contain memory areas specified in the given configuration. That means, if any area is not specified or has the wrong access-permissions, it will not be accessible, neither for the Linux root cell, not for any other cell. The same is applied to the I/O devices of the system: with a few possible exceptions, only devices listed in the configuration will be mapped with the IOMMU and its interrupt remapping (one exceptions is for example the PM-timer).

Later on, when Jailhouse has returned to Linux as guest, and Linux is used to create other cells, every cell configuration will again state the exact memory areas and I/O devices it wants to use. These are then *removed* from Linux's EPT and IOMMU mappings (if they were present in the original configuration) and hence, become inaccessible for it.

Metrics About Jailhouse The size of the hypervisor components and the performance of the presented start procedure can be seen in 6.2 on page 96 and 6.4 on page 99.

3 Trusted Execution

At the end of the last chapter, it was shown how important and powerful a single piece of software — the hypervisor — can become. After Jailhouse starts, the whole running software system on the target machine depends on the function of it — even the operating system that executed before it. Even more, the envisioned guests with higher safety requirements than the original operating system will depend on it and on its capabilities to separate them from the other parts of the system. All is concentrated on the proper function of this small, well defined piece of software.

But as of now, there is no way to check whether the loaded hypervisor image is intact and the one that is expected (e.g.: by the user). The only check done is a comparison of a commonly known value in the header of the image. And while this is also often true of other software — operation systems, kernel modules of those, or normal software — the effect of a failure in a hypervisor like Jailhouse is amplified by the type of guests it shall make possible.

This principle can also be observed at other points during the runtime of a computer, and is central motivation for the topic *Trusted Computing* [PMP11] — how can users gain "*trust*" in the software running on a computer? And while this task covers more, one central problem is to ensure that the expected software was started correctly in the first place and how this can be proven to another party. This subtask is also called *Trusted Boot* or *Trusted Execution*.

This chapter will give an introduction to this topic and the techniques that are intended to be used to *execute* Jailhouse in a *trusted* way. A short, general overview over the envisioned solution will be presented in Section 3.1 on the following page. Section 3.2 on page 27 will define the necessary terms and describe the general techniques to implement *Trusted Boot* on a modern computer. Section 3.3 on page 32 will then introduce the *Trusted Platform Module* (abbr. *TPM*), a hardware extension that is used to support this task. And finally, in Section 3.4 on page 40, the Intel *Trusted Execution Technology* (abbr. *TXT*) will be introduced. This is the

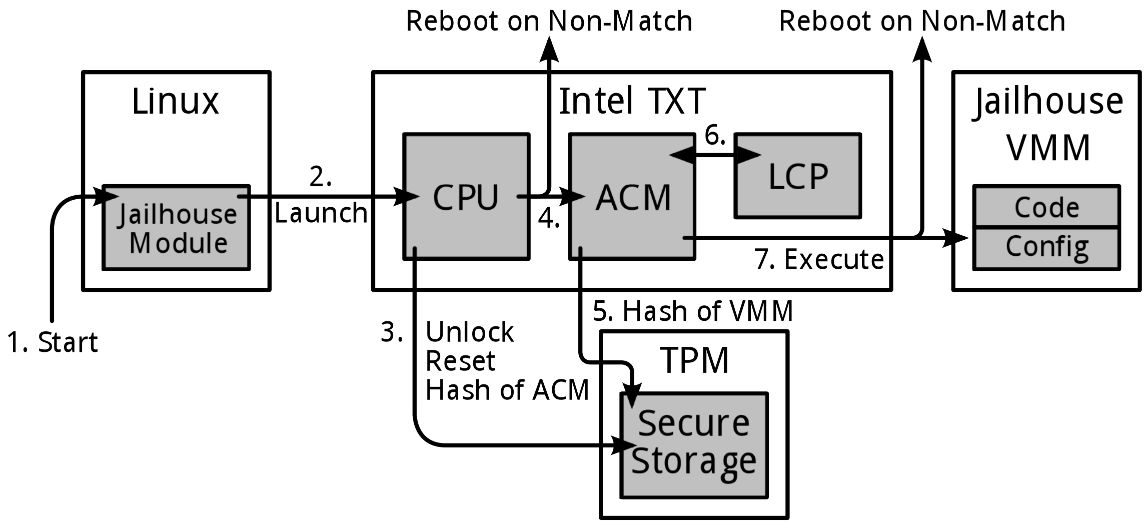

FIGURE 3.1: Rough overview over the process of executing the Jailhouse hypervisor in a trusted way, using Intel TXT and a TPM. All the necessary components will be explained in more detail throughout this chapter. A short explanation for the individual steps can be found in section 3.1.

central topic of this work and later it will be evaluate, how TXT can be combined with a hypervisor like Jailhouse.

3.1 Overview of the Envisioned Execution

Before going into the details of the trusted execution, this section will give a short overview of the envisioned solution for the hypervisor Jailhouse. The process is visualized in Figure 3.1:

1. The start of the process is the same as before: the kernel module receives the signal to start the hypervisor.
2. It will then decide to launch the hypervisor with the trusted execution and issue the appropriate commands.
3. After that, Intel TXT takes over control of the *whole* system. During this step, the CPU will execute a special procedure outside of user control. It will unlock special functions of the system's TPM (localities; explained later), reset the TPM's secure storage and save a hash of the following software in it (the ACM).
4. This saved hash is then compared, and the ACM (software supplied by Intel) only gets to execute if the hash matches an expected value of a cryptographic

signature (this signature is not alterable by the user). Otherwise the CPU will reset the system completely.

5. The ACM will then use the unlocked functions of the TPM to save a hash of Jailhouse in the secure storage.
6. (optionally) In case the user has (securely) specified in the TPM that LCPs shall be used (a form of hash lists), it becomes required to supply a list of accepted hashed to the ACM (the LCP). Using this, the ACM will search for a hash matching the one of Jailhouse that was just stored.
7. Should such a match be necessary and no hash be found, then the ACM will reset the complete system, too. In case no match is necessary, or in case one could be found, it will finally execute Jailhouse.

At each of the transitions (the 4. and 7.) some of the special functions of the TPM are disabled again (privileges, or more precise, localities are lowered). This means amongst other things, the stored hash values can not be reset anymore and any changes to them will show. Related to this, it is also possible for the TPM to unlock secrets based on those hashes — it will only be possible to access the secrets if the correct hashes are computed (this is called *sealing*).

The fact that this process has happened on a target machine can later be proven to other parties. In combination with the LCPs, this will make sure that only good hypervisors will launch.

3.2 Establishing Trust in Software

In order to describe how to establish "trust" in software, it is first necessarily to define the relevant terms and their relation to each other.

What is Trust in Software? The term "trust" itself is used in a variety of places and functions. A common definition, when used for software systems and thus also in this work, is given by the `RFC 4949` [Shi07]:

Definition 3.1. ***Trust:*** A feeling of certainty (sometimes based on inconclusive evidence) either (a) that the system will not fail or (b) that the system meets its specifications (i.e., the system does what it claims to do and does not perform unwanted functions).

If this definition is applied to the software system in this work — the Jailhouse hypervisor —, it can be narrowed down further. In order to trust the hypervisor's ability to manage its guests in the anticipated way, evidence is necessary that Jailhouse was correctly started and is in control of the virtualization features of the real hardware.

The definition also indicates that this "trust" is not necessarily based on warranties proven for the software system via means like formal verification. To distinguish this more explicitly, the same `RFC` also defines:

Definition 3.2. ***Trustworthy***: A system that not only is trusted, but also warrants that trust because the system's behavior can be validated in some convincing way, such as through formal analysis or code review.

So, when this work talks about *trusting a hypervisor*, or more explicitly *trusting in Jailhouse*, it states:

Definition 3.3. A third party *trusts* a system to *execute* Jailhouse when given convincing evidence that Jailhouse was successfully started and able to establish its control over the virtualization of the system's hardware.

This also implies that Jailhouse has to be trusted to protect itself from other influences, much like normal operating systems are expected to do the same. Whenever the system in question gives the convincing evidence in question, and the verifier judges it to be correct, then Jailhouse is trusted to be still in control.

But what constitutes such evidence and how can it be provided to a third party in a convincing way?

A common way for this is to provide the *code identity* of the system, and to give a convincing prove that this identity really belongs to the system in question [PMP11].

Describing the Code Identity of a System The state-of-the-art method to represent the *Code Identity* of a software is via a *Cryptographic Hash*, taken over the code-binary itself and all its inputs — for normal application this also means all libraries linked into it at runtime. Such a hash z, computed with a hash function $h(x)$ over the binary and inputs of a software x, is also called *measurement* in the context of this work.

Cryptographic hashes are chosen because they are *Collision Resistant* and *One-Way* [PP10]. With those properties it becomes infeasible hard to calculate a given code identity with another software than the one expected (intentional or unintentional) — it is a very good identity relation.

The best time to take such a measurement is before the software starts to execute. At this point, the hash will always be the same for the same code and input, even across different platforms. For software like an operation system or a hypervisor — software that takes over control of the platform — it is also important that the hash is taken by the software in control before them. Otherwise the measured software might influence the measurement or just lie about it. This in turn raises the question if the software taking the measurement is trusted as well, and to answer this, there has to be a measurement for this software as well.

The resulting sequence of measurements L is called *Chain of Trust* [TCG12]. The software currently in control of the system s_n is measured by its predecessor s_{n-1}, which in turn is measured by s_{n-2}. Ultimately though, this sequence has a starting point s_0, which no matter if it is hardware or software has to be trusted without any more evidence. This start-point is called the *Root of Trust*. If the chain is treated as proof, then the root of trust can be seen as axiom — always true.

An example for this procedure is given in Figure 3.2 on the next page. It also shows two possible applications of how the chain of trust and its measurements can be used: *Trusted Boot* and *Secure Boot* [TCG07]. Like established before, the final piece in such a chain has to be trusted to remain in control of the system and to protect itself from being taken over without an additional measurement being made. Depending on the software system, this may also mean that it has to measure every piece of software which can influence it in a untrusted way, so that it could change the trust decision of the trusting parties.

Usually, the involved hardware in this process is also treated as trusted without any further prof. Together with the measurements in L it builds the *Trusted Computing Base* (abbr. *TCB*) — all the code and hardware that needs to behave in the intended way.

But in order to convince a third party, may it be the user of the computer or an other computer system, that the hardware is currently executing the software system resulting from the shown chain of trust, it is also necessary to find a way to store this information and then to transmit it — both securely.

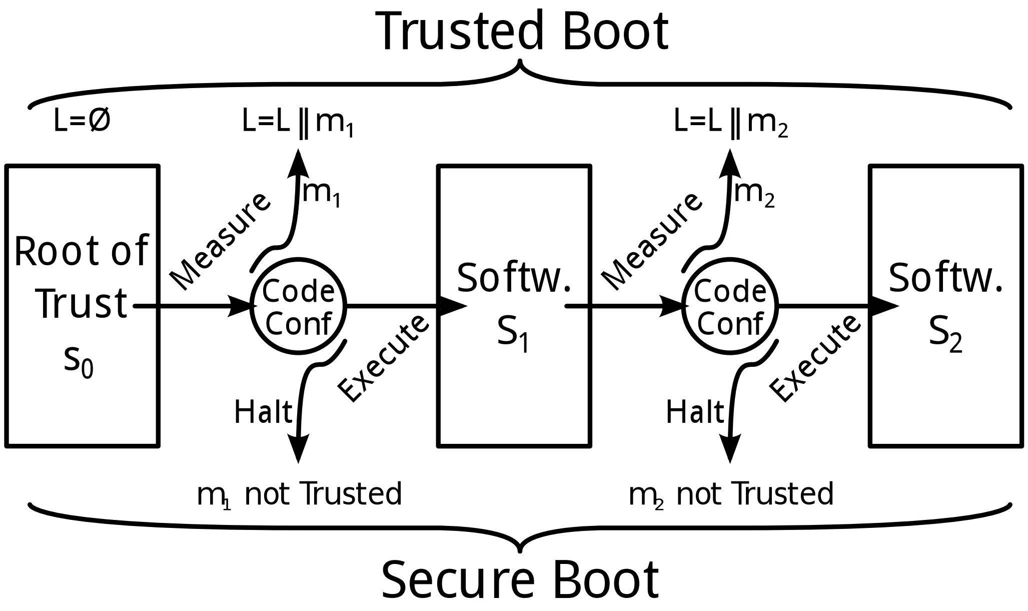

FIGURE 3.2: Diagram showing a chain of trust and how it is applied in two scenarios: *Trusted Boot* and *Secure Boot*. During trusted boot the measurements m_x are only appended into the chain L. With secure boot they are also compared to a reference value and if they don't match the system is halted.

Storing the Chain of Trust When storing the chain of trust L, it has to be made sure that it can't be influenced without being noticed. Any influence at all should be limited to appending new measurements.

Cryptographic hashes can again be used for a part of the problem. The relevant technique is called *Hash Chaining* and is also used in the TPM [TCG07] (later introduce in this chapter). With hash chains, no measurement is stored directly during the computations. Instead, a new measurement is first concatenated with the previous value, and then the result is hashed again. Only then the final value is stored into the "secure location".

Lets assume the root of trust starts with an initial value of

$$L = 0;$$

it then measures the next software with the result m_1. This value is not stored as such, but instead L is computed as

$$L = h(0 \parallel m_1);$$

and only then this value is stored. The measured software is then executed and repeats this, it measures the next software into m_2 and *extends* L again as

$$L = h(h(0 \parallel m_1) \parallel m_2).$$

This process is repeated till the chain of trust is complete.

Because L is a cryptographic hash, and thus one-way and collision resistant, it is infeasible to find any other combination of softwares, or values in general, that would result in the same hash-value. Not even when the software is execute in a different order:

$$m_1 \neq m_2 \implies h(h(0 \parallel m_1) \parallel m_2) \neq h(h(0 \parallel m_2) \parallel m_1).$$

But this still leaves open the problem of finding a "secure location" to store L, so it becomes impossible for any software to just overwrite it or assign a value directly during the hash chain. Both would destroy the chain.

Convincing a Third Party After storing L in a secure location, the next problem is to find a way to transmit it to other parties in a convincing way. In particular, the problem is to convince the other parties that the sent hash value L really belongs to the system in question and that it is the value computed and stored in the "secure location".

Convincing another person or system that a message is really from a expected partner is a common problem in cryptography. It is commonly solved by either sharing a secret that only the two communication partners know — symmetric cryptography —, or by asymmetric cryptography [PP10].

In case of asymmetric cryptography, the system in question would use a secret private key to sign the value L before sending both signature and value to the asking party. This system or person has to be in possession of the shared public key which belongs to the just used private key. Using this key, it can verify the received message. Because the private key is only possessed by the trusted party, the messages signed with it can be trusted to contain what is expected, and if they contain a value for L that matches a value of a trusted code identity, the system in question can be trusted to run the corresponding software. Any other outcome, may it be a wrong signature — made with an other private key — or a value for L that not corresponds to a trusted code identity, would lead to mistrust for the asked system.

But this leads to the question: how is the system in question supposed to store the private key, so it will be unknowable to anyone else?

This problem, along with finding the "secure location" to store L, is difficult to solve in software alone. While taking the measurements has been found possible to implement purely in software [GCK05, Spi00], secure storage is still an open issue [PMP11]. The solution is a hardware-based storage and will be introduced in the next section.

3.3 The Trusted Platform Module

The *Trusted Platform Module* (abbr. *TPM*) is a hardware module developed by the *Trusted Computing Group* (abbr. *TCG*) [TCG07, CYC+08, PMP11]. It builds the central piece in the *TCG*'s initiative to specify a common approach for trusted computing, with one target being to support trusted execution. The TPM itself is specified in a platform neutral way [TCG11a, TCG11b, TCG11c] and its implementation for the x86 architecture is specified on its own in [TCG12]. Most of the TPM enabled systems currently in the field are implementing the version 1.2 of the specification; the test system available for this work also ships a chip of this version — the further examination of the TPM is based on this version.

In the context of this work, the primary interest is focused on the TPM's ability to provide the "secure location" to save the chain of trust and its ability to generate verifiable messages from it, in the same manner as described in the previous section.

Hardware Integration of the TPM On x86 the TPM is integrated by connecting it via the *Low Pin Count* Bus (abbr. *LPC*; specified in [Int02]) to the chipset of the platform [TCG05]. In contrast to other I/O devices connected, it has no ability for DMA or any other direct influence over the behavior of the system — it is completely passive and only ever reacts on requests.

Because the TPM is designed to be inexpensive — one TPM chip costs about one dollar [Cor10] — it does not require much computation power from the chip itself. Timeouts to process single commands are specified from $750ms$ to up to $2s$ [TCG05]. The same holds true for the LPC Bus, which transmits a single byte in about $330ns$ — in comparison to the 20 year old 32 bit PCI bus, clocked with $33Mhz$, which

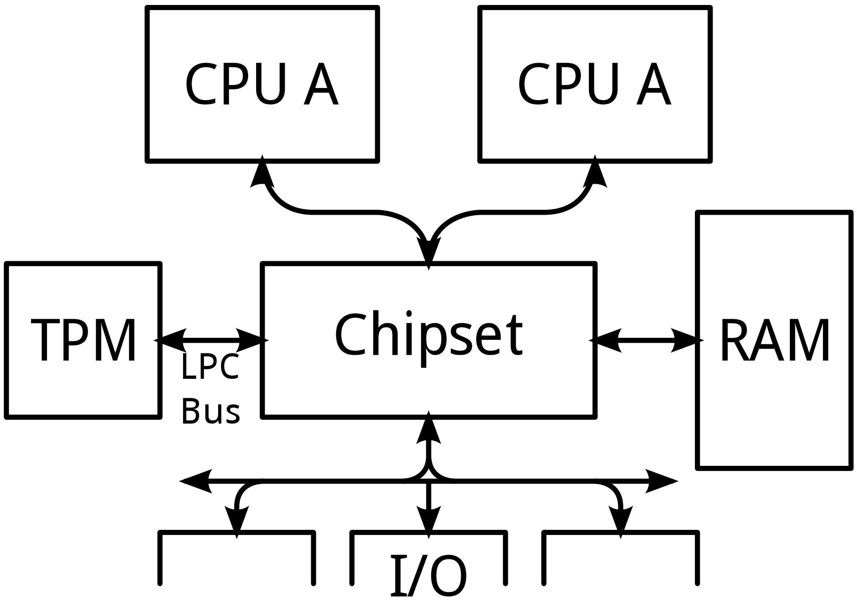

FIGURE 3.3: Overview over the integration of the TPM into the architecture of a x86 platform.

could transmit a byte in about 8ns, this is more than a magnitude slower. In general this means that the TPM can not be used to process big amounts of data.

The specification also requires some guarantees against hardware attacks on the TPM chip itself. It terms this as “tamper resistant”, more precisely this means that the chip has to be permanently connected to the platform — the motherboard —, so it can not be easily disassembled or transferred to another platform. Attempts of such tampering with the chip shall be evident upon inspection of it [TCG07].

Furthermore, it also requires the TPM packaging to limit attacks like pin probing, to gain knowledge of resident secrets, or electro magnetic scans, as form of side channel attacks to gain information of the functions executing on the TPM. Concerning the LPC bus, which was not designed exactly for this use, the specification only requires it to resist simple attacks and otherwise would need “expertise and possibly special hardware” [TCG11a].

Altogether, it is indicated that this should make it possible to certify a specific TPM implementation according to the FIPS 140–2 standard about cryptographic modules [FIP01], likely up to security level two. At the moment of writing, only one such chip is known and it was only evaluated for security level one [NIS14].

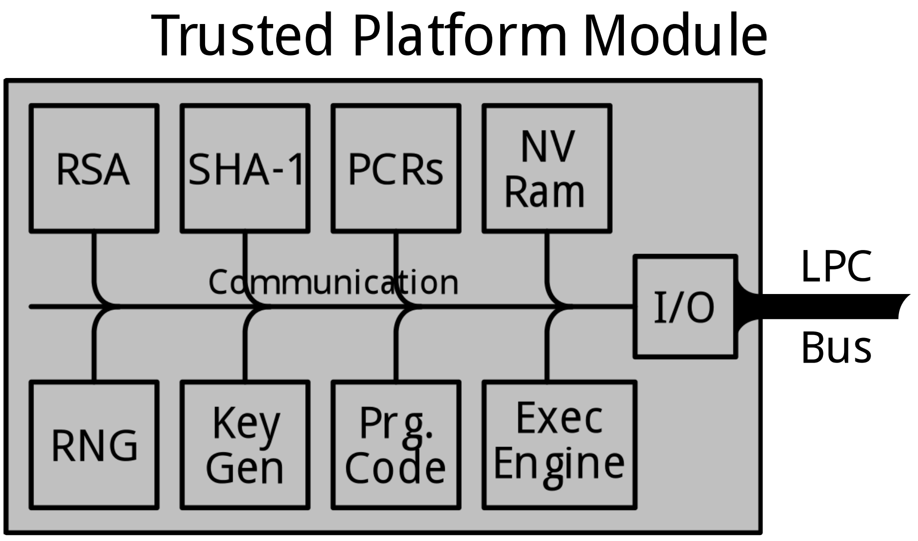

FIGURE 3.4: Overview of the components of a TPM chip.

3.3.1 Functional Overview

At the core of the functions, the TPM provides are three aspects: secure storage of keys, secure storage for *measurements* and support for trusted reporting. With those, it becomes finally possible to implement a chain of trust as described before. But those are not its only functions, and for this work in particular two more functions will become important: localities and NV RAM. A more detailed overview can be seen in Figure 3.4.

The depicted function blocks correspond to those specified in version 1.2. The implemented cryptographic algorithms in this version are limited to hashing, using the old SHA-1 algorithm (output size is 160 bit), asymmetric cryptography (including signing), using RSA with keys up to 2048 bits, and an own Random Number Generator with its own source for entropy. And although there is some non-volatile memory available on the chip, the specification only requires it to be 1280 byte big [TCG05].

Secure Storage of Measurements Like previously explained (3.2 on page 30), it is important to have a secure location to store measurements during the build of the chain of trust.

Because the TPM has only a very limited amount of storage available, it solves this problem with the *Hash Chaining* algorithm explained in the same section. To store the hashes, it provides 24 so called *Platform Configuration Registers* (abbr. *PCR*). These registers are volatile, but mostly only reset during platform resets (the registers with the indices 16 to 23 can be reset during the runtime, using the *locality* mechanism explained in 3.3.1 on page 38). Additionally, values can never be written directly, but they are always concatenated with the previous value and then hashed again — just as explained (the command is called `TPM_Extend`).

The PCRs with indices from zero to 15 (also called *static PCRs*) are used during the boot phase of the system (explained in 3.3.1 on page 37). They store information like the hash of the used BIOS, option ROMs, the used IPL, loaded OS, and for each the used configuration. Those can not be reset during the runtime and will reset during platform reset to zero. The remaining PCRs 16 to 23 (called *dynamic*) will be explained later.

Secure Storage for Keys The next problem is how a system can store private keys in a secure location, over a potentially very long time frame. Because of the limited amount of non-volatile RAM, storing them on the TPM directly is not a good option. Instead, the TPM is used to encrypt the private portion of such a asymmetric key or, in case of symmetric cryptography, the whole key. And once encrypted, the TPM will only decrypt them again in case a good authentication is given — this could be a pass phrase, a specific value stored in a set of PCRs, or the activation of a high enough locality. But again, for this operation it also needs a secret key (a RSA key in the case of the TPM)!

There are two (RSA-) keys permanently stored on the TPM: the *Endorsement Key* (abbr. *EK*) and the *Storage Root Key* (abbr. *SRK*).

The **EK** is stored on the TPM by the manufacturer and can never be changed. It is the TPM's identity and because the TPM is permanently connected to the platform, it is the identity of that, too. Because this intimate relationship can result in privacy issues, in case the public portion of the EK is used to permissive, its function is restricted to signing certificates of other keys (they are called AIK; further explanations will follow), stating: "this key belongs to the TPM with this EK". With the help of an external agency, the owner of the TPM can then request a second certificate for this new key, stating: "this key belongs to a trusted TPM"; and has thus decoupled the AIK from the EK. Ideally, there would also be a certificate stored on the TPM that attests that the EK belongs to a confirming TPM chip, signed by a similar agency, but this optional idea of the specification never prevailed [PMP11].

The **SRK** is created during the establishment of the TPM's ownership. Initially, the TPM is in an "unowned" state and the user has to first take ownership over it

(`TPM_TakeOwnership`). This process establishes a shared secret between the owner and the TPM (likely a password), and creates the SRK. Whenever the ownership is revoked (requires the secret) this key is removed as well.

The private parts of both keys never leave the TPM, not encrypted, and in no other circumstance, they are only used in operations on the TPM (the public parts can be retrieved).

When an other key is generated on the TPM, then its private part, together with other information, is encrypted with the public key of the SRK. Only then it is allowed to leave the TPM. This technique is called *binding* or *wrapping*. And because those bound keys are encrypted with the public part of the SRK and the corresponding private part never leaves the TPM, they only ever can be used inside that specific TPM[1] (because of this, the SRK is also called to *Root of Trust for Storage* (abbr. *RTS*)).

Next to the binding, it is also possible to specify additional criteria for decrypting or using such keys: it is possible to specify a password; it is possible to specify a set of PCRs, and only if those PCRs have the value, as at the time the key was created, the key becomes usable (also called *sealing*, security is based on the collision resistance of the hashes); and finally it is possible to specify a locality that has to be satisfied.

Next to normal RSA keys, the TPM also provides some that have a special meaning for it. One important kind are *Attestation Identity Keys* (abbr. *AIK*), RSA keys that are only used to sign information and with that, attest that those information belong to the TPM to which the AIK is bound. They are created and bound directly to the SRK of the TPM in question. The information that they are for this special purpose is encrypted along with the private part of the key. As an option, they can also be certified with the EK's certificate as parent (as described before), if there was one created for it[2].

Trusted Reporting With those two aspects solved, it is possible to construct a way to send convincing evidence to users or other computers about the code identity of

[1] It is also possible to create migratable keys or encrypt/decrypt arbitrary data, but this is out of context for this work.

[2] It is, of course, also possible to create a private certi cate chain, using the public key of the EK and the public keys of the AIKs, but for this work it is su cient to know the public portion of the key.

the platform. For this, the TPM provides a specific command: `TPM_Quote`; and calls this procedure *Remote Attestation*.

A *quote* is not much more than what is described in 3.2 on page 31. It is made up from a set of PCR values, a *nonce* that was provided to the challenged party by its challenger, and a signature of both which is created with the private portion of an AIK bound to the challenged TPM. The *nonce* is a random number. It is created by the challenger only just at the time he requests the remote attestation. Because it is included in the signature, the signature can not be used for any other request (they will have other nonces) — avoiding replay attacks.

This is all done *on* the TPM, and afterwards the challenged party sends the result back to the challenger. He in turn can then verify the quote by testing the signature with the *known* public portion of the AIK (as with other asymmetric crypto algorithms, the public portion has to be obtained in a secure way), and examine the set of PCRs values, if they match a trusted set of measurements.

Because the AIK is bound by the SRK of the TPM (its private portion is encrypted with it), and the SRK never leaves the TPM (not even encrypted), it is infeasible complex to forge the signature of the PCR values and the nonce outside of the TPM. This means, the operation of taking the PCR values and making the signature have to happen on the TPM to which the AIK belongs. And by knowing the public portion of this AIK, the verifier can impeccable determine to which TPM the signature belongs — or if it belongs to a TPM at all.

Any other outcome — a invalid signature, the wrong AIK, a set of PCR values not matching a valid set of measurements, or even just a timeout — can then be treated as strong sign to mistrust the challenged system.

This mechanism, along with the AIK/EK-Certificates, is also called the *Root of Trust for Reporting* (abbr. *RTR*).

Chain of Trust using the TPM Next to these 3 main features of the TPM itself, the TCG also defines two possible ways to create a convincing chain of trust (see 3.2 on page 28): a static [TCG12] and a dynamic [TCG13] chain of trust.

Because Intel TXT is one implementation of the dynamic chain of trust, its explanation is deferred to 3.4 on page 40. Here, only the static will be explained. An overview of how it works can be seen in Figure 3.5 on the next page.

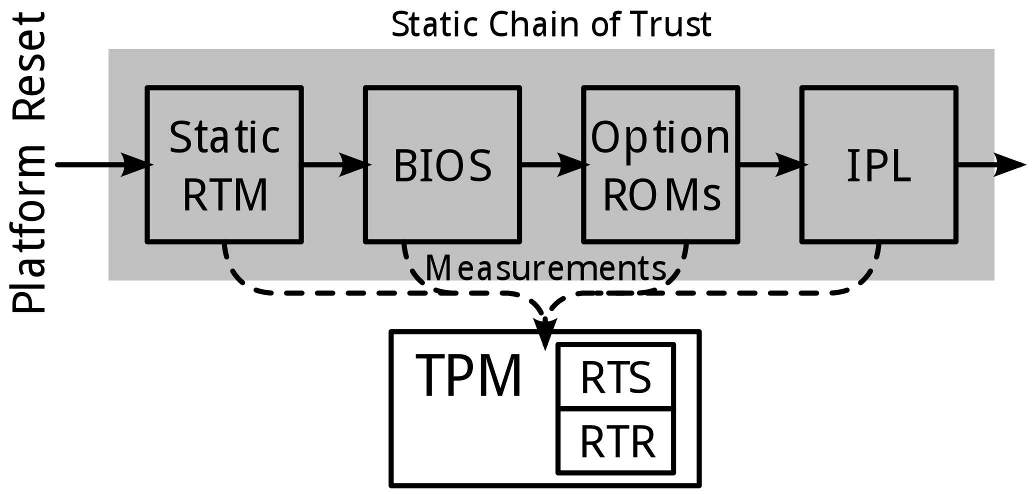

Figure 3.5: Diagram showing the ow of measurements and software execution during the static chain of trust. The chain is started by a special, embedded piece of software, the *Static Root of Trust of Measurement* that is placed directly at the reset vector of the system.

The root of trust in this chain (also called the *Static Root of Trust of Measurement* (abbr. *RTM*)) is a small piece of software directly loaded after the platform reset/start. At this point the platform is *trusted* to be in a good state. This software starts up the TPM and measures the BIOS and its configuration into PCR zero (it *may* also measure itself). After that, it relinquishes control to the BIOS, which in turn measures any optional ROM with the corresponding configuration. This chain is continued unbroken till the IPL has measured and launched an operation system (this may also measure applications, but is not required to).

With this chain, it can, for example, be proven to the user that the system was in a good state during the boot and loaded the expected operation system. With the assumption that the OS can protect itself properly, the user can thus trust the system. But it also means that all components during that chain become part of the TCB and have to be trusted. With modern OS kernels reaching into tens of millions lines of code [DWQM14], this becomes is a considerably large amount, coming from a big variety of sources.

This is one of the main motivations for the use of a dynamic root of trust, as explained later.

Localities To better support the dynamic chain of trust, TPM 1.2 introduced the concept of *localities*, five over all (0–4) [TCG12]. They can also be seen as different levels of privileges when talking to the TPM. And as mentioned before, the TPM allows localities as one kind of authorization criteria — keys, NV RAM and other commands can be set the require a certain locality to be used.

The highest locality (4) can only be used by trusted hardware — never software — and has the most privileges. It can, for example, use this privilege to reset certain PCRs (16–20 and 23). To protect it from software, the specification [TCG05] requires the hardware manufacturer to implement explicit security measures — it may be done by adding special bus cycles to the LPC bus. But the exact mechanism is not specified and could not be found[3], the manufacturer thus have to be trusted to implement this securely to be compliant to the specification.

The localities three, two and one are to be used by software of different trustworthiness, but neither is specified what this exactly means, nor how those are protected. It is up to the implementation to specify this — it will later be shown how TXT manages this issue.

The remaining locality zero is used by the static chain of trust and legacy applications.

NV RAM The last TPM element, this work makes use of, is the *Non-Volatile Ram* (abbr. *NV RAM*). Although not much (at least 1280 Byte), it can be used to save and protect a few information permanently on the platform.

To use the RAM, the owner — and only the owner — has to define areas in it (also called indices). And during that, he can decide, similar to the keys, what authorization it requires to write or read from the defined area (with the same types as with the keys: (owner) password, PCR state, locality). Afterwards, everyone possessing the correct authorization can use the RAM at will.

Like the other parts of the TPM, it is accessed through the LPC interface and thus quite slow. And additionally to the already scarce space required as minimum in total, the specifications also requires some predefined areas for other purposes. This reduces the required minimum space for the user to 512 byte [TCG12], and therefore requires good care about what information are to be stored in it.

[3]Upon personal enquiry to Intel, it was said that this information is con dential.

3.4 Intel's Trusted Execution Technologies

Till now it was shown how it is possible to implement a chain of trust with the help of the TPM and the support in the variety of firmwares of the specific platform. This *static* chain of trust stems from the root of trust in the hardware itself — it is assumed to be in a good state after reset — and the small piece of software directly loaded at each reset — the static RTM. This RTM is specified to be immutable [TCG12], and it will measure every unintended change in the following code into one of the TPM's PCRs. Hence, if the hardware and the RTM is trusted, then this state is a good starting point for the chain, and if this chain remains unbroken and with only expected measurements, till it is in the desired environment (for example, an OS or a hypervisor), then the system in question can be trusted.

But this requires to trust every piece of software that contributed up to this final point — the TCB — and this might include code in the size of multiple tens of millions lines. In addition, it requires support for it in every piece along the way, or it might miss critical measurements.

Those two points are the major motivations for the *dynamic* chain of trust. In this scheme the chain doesn't start at boot time — although the static RTM still does its measurements as intended, as said, it is immutable —, but instead, it can be started at any time during the operations of the system. The main obstacle for this is, as explained in 3.2 on page 28, each piece of software ran before the new root of trust can potentially influence it. In this new situation, even the hardware can influence it, for example, by unintended DMA accesses — the exact hardware state can not be known beforehand. The *DRTM* (*Dynamic RTM*) needs a similar known-good state of the hardware as the static one, to start operations [TCG13].

This led to the implementation of processor extensions on both platforms: Intel and AMD. AMD's implementation [AMD13, AMD05] has been introduced together with its extension for hardware virtualization and is also documented alongside it. In this work however, the focus is on Intel's implementation TXT [Int14c, FG13] — both share the same concepts, but are implemented differently, and at the time this work was started, Jailhouse didn't have fully functional AMD support.

This section will give an overview of the components involved in TXT and how it enables the environment for the DRTM. Details about the resulting requirements for a software that wants to implement support for TXT will be given in Chapter 5 on page 61.

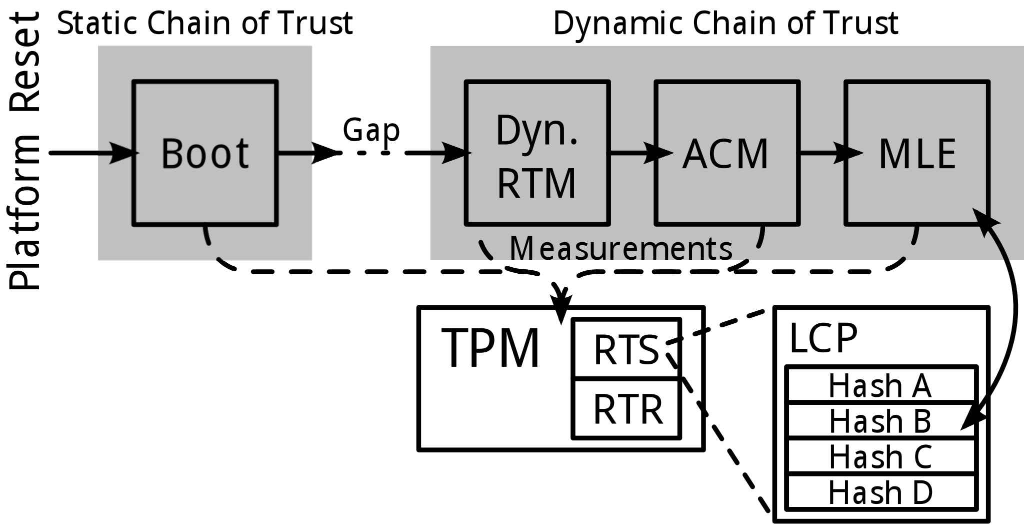

Figure 3.6: Overview over the software components involved in Intel's dynamic chain of trust. The \Gap" can represent any software running after the boot and before the DRTM.

3.4.1 Overview

The components involved and the resulting chain of trust can be seen in figure 3.6:

- **Hardware**: TXT is an optional processor and chipset extension (it existence can be checked via `CPUID` [Int14a]). In cooperation with the platform's IOMMU (Intel's VT-d) and TPM, it is responsible to initialize the known-good state to start the DRTM and to take the first measurement.
- **Static Chain**: The measurements that are made after the system reset by the static RTM are still made and can also be used in subsequent Quotes.
- **LCP**: The *Launch Control Policies* are a list of rules in a format also invented by Intel [Int14c, FG13] (simplified, it is a list of hashes). It is possible with them to enforce different checks on the system *in the ACM*: in case this feature is activated, only a MLE whose measurement matches one stored in the LCP can be started, any other would cause a system reset. The information whether it *is* activated is stored securely in the TPM's NV RAM (more details will be presented in 3.4.3 on page 48).
- **ACM**: The *Authenticated Code Module* is a small firmware that is provided by Intel or the platform's manufacturer (it is also frequently called *SINIT*). It is the first software called after the DRTM has established its environment, and it is responsible to check the setup of the MLE, measure it (depending on the LCP, enforce rules) and ultimately jump to the entry point of the *MLE*.

- **MLE**: The *Measured Launched Environment* is the target software and its configuration. It is launched by the ACM, and it is the first piece of software that is provided by the system's user. More details about the target MLE in this work will be given in Chapter 5 on page 61.

This list also summarizes the whole TCB of a successful ***measured launch*** (starting TXT and entering the MLE). It does not include any components ran in the ***Gap*** before the launch, components of the static chain are optional, and components running after the launch are not included, as long as the MLE isolates itself properly.

That means, if this technique can be used with a hypervisor — or any other software for that matter —, it is only necessary to trust the hypervisor itself, the ACM and the involved hardware (including the TPM). Out of those, hardware and TPM are also included in the static chain and the ACM can be compared to the initial software run after the system reset. Compared to the static chain, that in this work would also include the whole operation system, this reduces the TCB by multiple million lines of code and emphasizes the trust mainly on the launched hypervisor and its functionality.

3.4.2 Intel's Safe Mode Extension

In order to start this new dynamic chain, the system has to establish a known-good state in which the hard- and software can be trusted, and in which measurements and the measured software can not be influenced.

This is done with the new instruction `GETSEC`, introduced with TXT into Intel's ISA [Int14a]. `GETSEC` is part of the *Safer Mode Extensions* (abbr. *SMX*). It implements the processor-related features of TXT.

Although starting the measured launch is the major functionality of `GETSEC`, it also implements other functions. Those are selected via a value put into the `EAX` register before calling it and they are currently made up from 8 different sub-functions. For the use in this work, the following are the most important:

0. `CAPABILITIES` is similar to `CPUID` and returns bit vectors containing bits set for TXT functions that are available on the system.

4. `SENTER` starts the measured launch and is the main functionality at this point. A more detailed description can be found below.

5. `SEXIT` ends a measured environment previously erected by a call to `GETSEC[SENTER]`. This includes re-enabling events that have been masked before and not already unmasked in the MLE, and it closes the private TXT config area (see 6.1 on page 76).

6. `PARAMETERS` is used to retrieve information about the different parameters of the TXT implementation on the questioned processor (how big can the AC module be, what versions of it are supported, which memory caching type can be used for it and more; at the moment there are only a few parameters for TXT itself).

7. `SMCTRL` controls the measured environment from within the MLE. Currently this is only used to re-enable SMIs after the launch.

8. `WAKEUP` wakes sleeping processors after they have been halted for the measured launch (explained below).

Measured Launch with `SENTER` The measured launch can only be done with the *Bootstrap Processor* (abbr. *BSP*, often also called *ILP*). This is the processor that was first activated after the system reset and initiated the remaining system, including the wakeup of the other processors (also called the *Responding Logical Processors* (abbr. *RLP*)).

Once the system has loaded the ACM into the correct memory position and done the other necessary setup steps (see 6.3 on page 80), the BSP can make the `GETSEC[SENTER]` call, together with the physical address of the ACM and its size as parameters.

First off all, `GETSEC` will check several states of the system to see if the call was legal. This includes basic checks, such as, if SMX is unavailable or not enabled, if another `SENTER` is already running and no yet terminated (with `SEXIT`), or if the call was made from within the VMX non-root operations. Furthermore, it will check if the system, especially the chipset, has all the necessary components, such as TXT-capabilities and a built-in TPM. Should any of these checks fail at this early state, then the call will return to the calling software with an exception. Errors later during the launch will always result in a system reset (also referred to as *TXT-reset*).

After these preliminary checks, the processor will rendezvous all other available processors in the system and disable all but the BSP (this goes as far, as to disable their capabilities to do any memory or I/O transactions). This is done regardless in what state the RLPs were before the BSP called `GETSEC`, and they can only be woken again later in the MLE. At this point only the BSP is still awake.

The BSP will now disable all external events, including the `INIT#`, `A20M`, `NMI#` and `SMI#` pins, but also any normal IRQ (via the `IF` flag in the `EFLAGS` register). To prevent information about the execution to leak, it also disables all debugging and performance counting features of the processor and clears their content. These are typical side channels that can be used to analyse the systems behavior without direct access at the time of the measured launch.

Before the launch, the ACM was loaded into a reserved memory area called the *DMA Protected Range* (abbr. *DPR*) by the MLE setup (see 6.3 on page 80). This area is allocated and locked in place by the BIOS and is handled specially by the IOMMU: each memory access from an external device to this area is blocked. In addition to this, the BSP will load the ACM from this area into an internal *authenticated code execution area*. In this area, it is secured from all other processors and external devices. This guarantee is given by the Intel Manual [Int14a], and the TXT manual [Int14c] furthermore states that this area is located within the processor.

Because the processor is not specified to test the DPR before it measures the ACM, it is important that this guarantee is held. Otherwise, it might be possible to influence the ACM via DMA after it was already measured, because the DPR was not configured correctly or the ACM was not placed inside this range. In order to later trust the whole system, the processor has to be trusted in this. And because all other processors are disabled, the only source of changes to this area can now come from the BSP itself. This completes the save environment for the measured launch, an overview over the resulting architecture can be seen in Figure 3.7 on the facing page.

Using this environment, the BSP will now check if the loaded ACM has a valid signature. For this, it will use a (RSA-) signature saved in the ACM binary itself and a public key known to the processor (through the TXT capable chipset). Only if the signature signs the loaded module and is made with the private key corresponding to the public key known to the processor, `GETSEC` will continue the execution. Unless the private key Intel uses to sign their ACMs is disclosed, this will prevent any other binary from acting as an ACM.

Furthermore, the BSP will now activate locality 4 of the attached TPM, reset the PCRs 17–22, extend the hash of the checked ACM into PCR 17 and close the locality

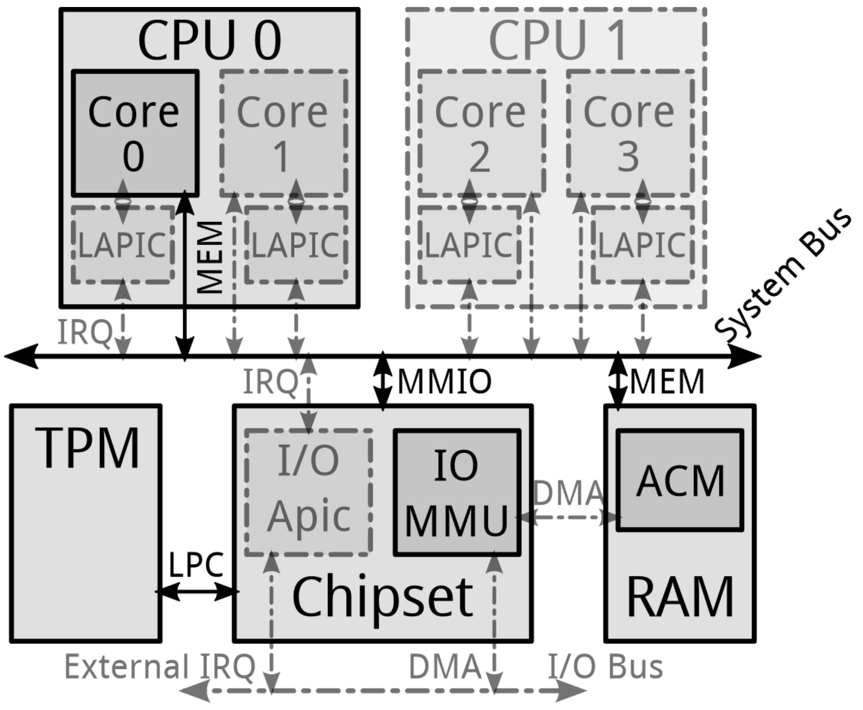

FIGURE 3.7: Diagram showing the state of execution on a system after `GETSEC[SENTER]` has erected its save environment for the measured launch. Only those parts painted with drawn through, black lines are still active and able to interact with the remaining processor and the memory.

4 again. Additionally to the signature check, this can later be used in a TPM Quote during a remote attestation to attest that the ACM was really executed.

Finally, after this *measurement* is done, `GETSEC` will bring the processor into a state as documented in [Int14a], open locality 3 and start executing the loaded and authenticated ACM.

The Authenticated Code Module Details about the ACM itself, internal details about its function, are not specified. Because it is closed source, only the effects of it can be known at this point.

From the different error states specified in the TXT specification and the data given to the MLE via the TXT heap (see 6.1 on page 77), it can be derived that it does extensive checks on the MLE-setup (done before the call of `GETSEC`), that it collects information about the memory and processor layout of the system (ACPI's MADT), and that it provides these information in a validated form to the MLE — so it doesn't have to use unvalidated data from the BIOS or other firmwares. Additionally, it can be configured to enforce rules in form of the LCPs onto the system (see 3.4.3 on page 48).

Any failure in those checks will result in an immediate system reset to prevent compromises — the system reset will also reset the TPM's PCRs.

Finally, it will measure the configured MLE, extend this hash into either PCR 17 or 18 (depending on a configuration option the MLE-setup can specify), apply the processor state described in 3.4.2 on the next page, close locality 3, open locality 2, and jump into the MLE code. From this point on, the MLE will be in control of the system and has to bring up the system from the applied state into its own defined work environment. Those issues will be discussed in Chapter 5 on page 61.

How the ACM Measured the MLE In contrast to the measurement done by `GETSEC`, the measurement done by the ACM is not specified by physical addresses — it will not use a physical address as start point and a size in bytes to hash a physical continues memory space.

Instead, it will use a page table provided to it by whatever code does the setup before the invocation of `GETSEC[SENTER]` (this table is also frequently called the *MLE page table*). To specify what address range in this table represents the MLE, it will use a **linear** address as start and another linear address as end address. Both are specified in the *MLE header* from *within* the MLE (see [Int14c] for a full spec.), next to another linear address used as *entry point* for the final jump. By taking these information from within the MLE, they are measured as well and can thus not be changed unnoticed.

As general format for the page table a regular *PAE* page table layout is used. It is however restricted by a set of rules, which are specified to be checked by the ACM during the launch [Int14c]. The most important for this work are the following (refer to the spec. for the full list):

- it may only contain 4 KB pages;
- all linear and physical addresses must be below 4 GB;
- when using a breadth-first search on the page table, it must produce only increasing physical addresses;
- after the first valid page, there may *not* be any invalid pages till the end of the MLE — there may not be any gaps in the (linear) mapping of the MLE.

Both, the first valid page of the MLE and its end, are taken directly from the linear addresses stored in its header. The process of the measurement itself is visualized

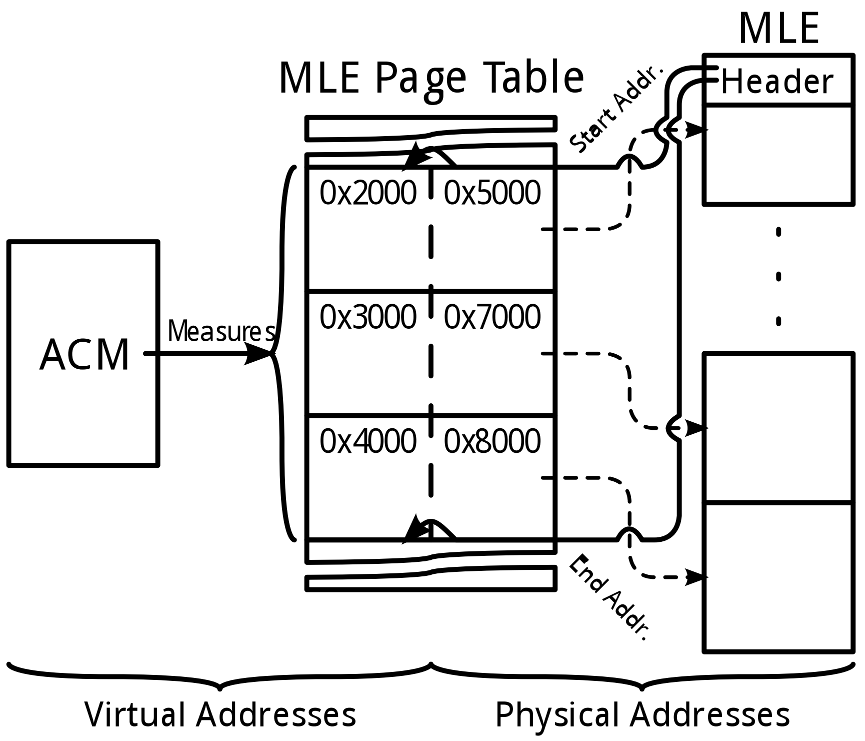

Figure 3.8: Overview of how the ACM utilizes the MLE page table during the measurement of the MLE.

in figure 3.8 — it will lookup the first valid page and then measure every page till it finds the end address.

By deploying this scheme, it is possible to distribute the MLE over different areas of the physical space. In case of large MLEs, this solves the problem of finding a large enough continue physical location for it. Because Jailhouse already requires a reserved, continues physical space before it starts, this table would not be necessary for the implementation in this work.

The Common Processor State Set by the ACM As the last action of the ACM, before it yields control to the MLE, it will apply a common processor state to the BSP. By doing so, it creates a known baseline that can not be influenced from the outside — not by any legitimate setup, but also not by any attacker. Afterwards, it is the MLE's task to work with this state, or to bring it back into a suitable state for itself, but it can always trust the base state.

The most important changes that are applied are listed in Table 3.9 on the following page.

It is important to note, that this also implies a change to the *unpaged protected mode*

Resource	ILP / BSP
CR0	PG = 0, AM = 0, WP = 0, CD = 0, NW = 0, NE = 1, PE = 1, others unchanged
CR4	`00004000h`
EFLAGS	`000000XXh` (`XX` = undefined)
EIP, EBX	the address of the first instruction in the MLE (this address is listed as linear address in the MLE's header, along with the start and size of the MLE)
ESP, EBP, EAX, EDI, ESI	undefined
ECX	pointer to the MLE page table
CS	base = 0, limit = 4GB, executable code
DS, ES, SS	undefined
GDTR	unchanged from ACM
IA32_EFER	0

FIGURE 3.9: Table showing the changed architectural state of the processor after the ACM gives control over the system to the MLE. This table lists not every change, a complete version may be found in [Int14c].

(32 bit), no matter in what operating mode the system was before the measured launch. Furthermore, there is *no* valid data segment set, hence it is not possible to write anywhere in this environment.

3.4.3 Controlling the Launch of Software with the DRTM

Implementing support for this measured launch will make it possible to initiate a *trusted boot* from a secure hardware state, at any given point during the lifetime of a system — measurements of each component are stored, but no rules are enforced. This is also depicted in Figure 3.2 on page 30 (at the top). After this procedure, the user of the system or a remote machine can verify the result of the launch with the use of *remote attestation* (see 3.3.1 on page 36).

In addition to this, TXT also supports the scheme of *secure boot* — the measurements are not only taken and stored, but also compared to a reference value (depicted in the same figure as before, but at the bottom). When TXT is used on the real hardware (compare to the opposite in 7.5.1 on page 113), this scheme can enforce

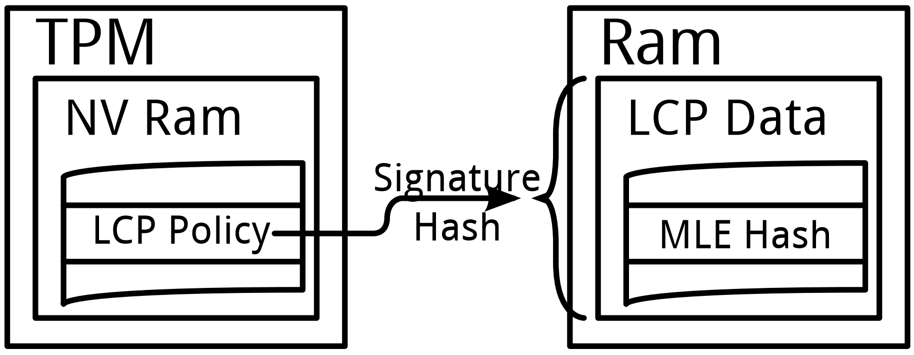

FIGURE 3.10: Overview over the storage concept of LCPs.

that only correct setups of the MLE can ever be launched with `GETSEC`, and hence get access to locality 2 of the TPM (along with other methods, this may be used as authorization for secrets). If no match is found for the measured MLE, then the system is reset by the ACM.

Whether “only” the trusted boot is used, or the secure boot, is decided by a small structure stored within the NV RAM of the system’s TPM. By specifying a strict authorization for this index of the RAM, it becomes impossible to change this decision as unauthorized user.

The mechanism realising this are the so called *Launch Control Policies* (abbr. *LCP*) [Int14c]. They are split into two part: one is the just introduced part stored in the NV RAM, and the other is, simply put, a list of acceptable hashes for different components of the measured system.

LCP Storage The basic storage scheme can be seen in Figure 3.10. It is made up from two parts: the *Policy* and the *Data*.

The LCP policy is a structure 54 bytes big and is required regardless of the system doing a secure or trusted boot — it is the part deciding whichever scheme is done. Next to this decision, it also contains fields for version requirements on the ACM, what hash algorithm should be used (as with the TPM 1.2, it only supports SHA-1) and most importantly, the hash of the *data* part of the LCP (if no secure boot is done, this can be left blank).

Like explained in 3.3.1 on page 39, the index to store the policy has to be created by the owner. In doing so, he also can decide on the access privileges for the index. There are no exact requirements for the privileges. But to secure the data part — the hash of it —, it should not be writeable by anyone else. One example could be:

Element Type	Description
MLE	Contains a known-good hash of a MLE.
PCONF	A list of accepted PCR values. This may be used to specify acceptable outcomes from the static chain that are stored in the PCRs 0–15.
SBIOS	Can be used to store hashes of BIOS images, as they would be measured by the static chain. But this is not specified any further, and as long as the user has no intimate knowledge of this value from his BIOS manufacturer it is not usable.
Custom	May be used by the MLE later during the measured launch and can be made up from any data the MLE designer sees fit.

Figure 3.11: An overview over the possible hashes that can be stored in a LCP data list.

writeable only by the owner, readable only with locality three and two (ACM and MLE).

The LCP data is made up from multiple lists of acceptable hash combinations. This is not only limited to hashes of the MLE, but can also contain hashes of other system parts that the ACM will check upon (an overview can be seen in Table 3.11). Other than that, it can also contain a signature (and the corresponding public key) made by the issuer of the LCP.

At runtime, the LCP data has to be loaded into the main memory before issuing `GETSEC` and starting the measured launch.

Securing the LCP Data To secure that these data are not manipulated before used by the ACM, they are bound to the TPM. This done by storing a measurement over all hash lists in the LCP policy part.

Such a measurement is a hash of the concatenated individual measurements of each stored list. These, in turn, are either just a hash of the list, or, in case the list contains a signature, it is the hash of the public key whose private counterpart made the signature. This makes it possible to exchange a list without alternating the LCP policy, as long as the issuer is in possession of the private key to make the proper signature.

Gained security? However, we will later see (in 7.5.1 on page 113) that the LCPs do not add much security to the system; in essence, it still is required to do a *remote* *attestation* to *prove* the system's state.

They acts as gate keeper if the measured launch is done on the real hardware with the proper instructions. In this case, they also adds flexibility, and because of the signature mechanism, they are not complex to update and to maintain.

In contrast to this, AMD doesn't provide any such feature in its own implementation of the dynamic chain of trust [AMD13, AMD05]. This implementation does only provide the known-good hardware state and the measurement of the MLE. All other features have to be implemented inside the source for the MLE.

4 Related Work

Trusted computing and virtualization are very big field with a variety of applications. The principles of their working were shown in Chapter 2 on page 5 and 3 on page 25. Additionally, it was introduced in 3.4 on page 40, how Intel TXT can improve the use of these technologies in a software system. Later on, the requirements for designing and implementing a solution using these improvements will be shown.

But next to the application in this work, to implement Intel TXT for a hypervisor like Jailhouse (see 2.3 on page 15), there are also other projects that want to make use of TXT.

In this chapter, some of the most prominent examples of these projects will be introduced shortly, and it will be shown how they differ from the approach in this work. The start will make *TBoot* in Section 4.1, Intel's reference implementation of TXT. It provides an alternative way to boot a classic operating system, with the added possibilities provided by the dynamic chain of trust. Following that, *Flicker* will be introduce in Section 4.2 on page 55, a framework to execute small specialised applications from within a running operating system and the added potential to remote attest their execution. In Section 4.3 on page 57, an improvement upon this concept, the hypervisor *TrustVisor*, will be presented. It follows the same idea of Flicker, but with a hypervisor started directly after the boot process (in contrast to Jailhouse). And finally, in Section 4.4 on page 58, this chapter will finish by giving some extra directions for material concerning trusted computing.

4.1 Trusted Boot with TBoot

TBoot [FG13, tbo] can be seen as an extended IPL and is Intel's reference implementation for TXT. Its basic architecture can be seen in Figure 4.1 on the following page.

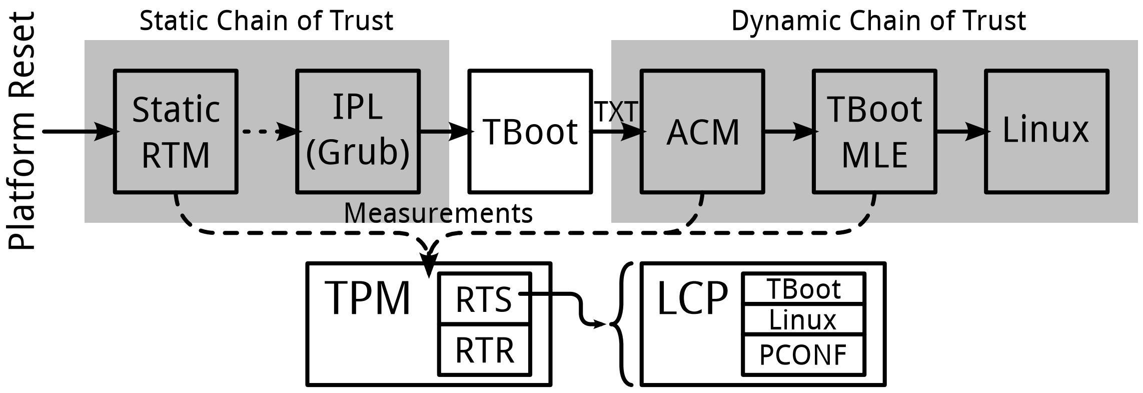

FIGURE 4.1: Diagram showing the start of Linux with the support of TBoot.

TBoot is loaded directly by the system's IPL (any boot loader that supports the multiboot specification [OFBI10]), at any 32 bit address above one megabyte (derived from the ELF header of the image). Any argument that it needs, for example command line options, the system's ACM and the Linux kernel image, are loaded alongside the image[1]. After this, the IPL will launch TBoot. This happens after the static chain has finished and thus the IPL doesn't have to support TPM — for example, any unmodified Grub [Oku12] will do.

TBoot will then launch the dynamic chain by doing the proper setup and calling `GETSEC[SENTER]` with the loaded ACM as argument. The MLE though is not yet the Linux kernel, this would need several modifications to its loading process (as will be shown later). Instead, the TBoot image itself is also the MLE that will be launched by the ACM, once it has finished. This step makes it also possible to include TBoot itself in the applied LCPs and thus include it in the system's TCB. Other than that, the LCPs may also contain any other element that is listed in Table 3.11 on page 50.

One custom element TBoot specifies and adds is the Linux kernel image. Once TBoot has started and done its setup in the trusted environment, it will go on and look up this custom element. If it finds any, it will compare the given Linux kernel image with this element's hash and proceed only if they match.

The final step after this is to extend the kernel's measurement into PCR 19 and start the kernel.

Afterwards, during the runtime of the launched Linux, the user can make use of any available TPM software stack to implement the remote attestation and use the

[1]Both loads are part of the multiboot speci cation, which makes it also possible to start without any 16 bit code.

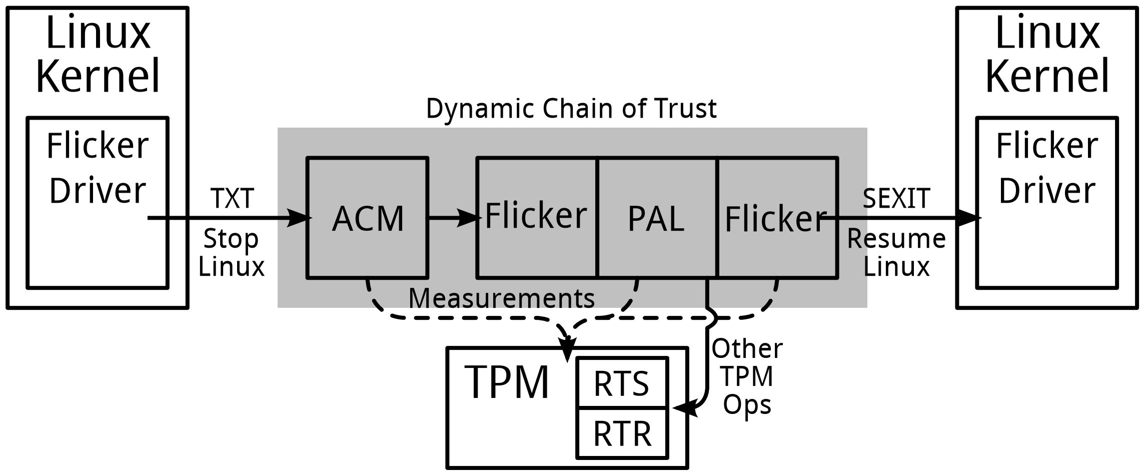

FIGURE 4.2: Diagram showing a Flicker session. The Linux that started the session is completely disabled during the dynamic chain of trust and will only be re-enabled once Flicker has nished.

measured values, made during the just explained dynamic chain of trust.

4.2 Running Small Applications in a Trusted Environment with Flicker

Flicker [MPP+08] takes a completely different approach to use the new dynamic chains. It was one of the first projects that made practical use of TXT and its counterpart from AMD. The target is not to gain trust in a system controlling software, like an operating system or a hypervisor, but to gain trust in a small application — also called a *PAL* (*Piece of Application Logic*) —, the advantage being, that only trust for the small TCB is necessary in order to trust the application.

A sample session of this can be seen in Figure 4.2. Flicker is started during the execution of a normal operations system (it supports Windows and Linux). Because it makes use of the SMX operations, it requires a small driver-like part in the operating system's kernel to gain the proper privileges. This driver will allocate the memory and load the target Flicker image and the system's ACM. Once it has done all the other necessary setup steps it will start the session.

The Flicker image is constructed from an initial setup part (the core) and the PAL. While the core is always 32 bit x86 code, the PAL can be anything, but has to start with 32 bit and do any further setup itself. At runtime the two interact with each other through call gates — the PAL runs at ring 3.

Starting with the session, the driver disables all but one processor core (the BSP) and saves their execution state into a private area. It then does the appropriate call to start the dynamic chain of trust. This will cause the measurement of the ACM, and following that, the measurement of the Flicker image into PCR 17 and 18; and once done, the control is given to the Flicker core — it does not make use of Intel's LCPs.

The core does only do a minimal setup. It does not rely on virtualization to protect itself, but it uses segmentation and x86's privilege levels to encapsulate the PAL into ring 3 before executing it. During this execution, the PAL can do everything that its privilege level allows and additionally it can also make use of the TPM in any way it sees fit. Once finished, it jumps back to the core through a call gate, and this will finish up the session by extending a well-known value into PCR 17.

Only then it returns control to the Flicker driver and this will restart the previously stopped processors and restore their original states. That also means that the original OS is completely stopped during the session.

Using remote attestation, the user can then prove that the PAL has run and, depending on the PAL's application logic, other properties about the finished run — the PAL may decide to extend values to other PCRs. The final extend of the well-known value will prove that the PAL has finished running — PCR 17 can only be extended in locality 4–2, and these are only active during the dynamic chain [TCG05]. Any sensitive data of the PAL can be secured by the TPM's sealing and binding features, but this is the PAL's responsibility.

While Flicker reduces the TCB of the application in question drastically (it will only contain the Flicker core, the PAL and the provided ACM and hardware for TXT), it also imposes some drastic restriction. Like explained, the normal operations of the system are completely blocked during the session, and the setup and launch have to be redone for each run. It will later be shown in 6.8 on page 97 that this adds some considerable overhead. Additionally, the current implementation is only able to run on 32 bit operating systems, reducing the field of application further.

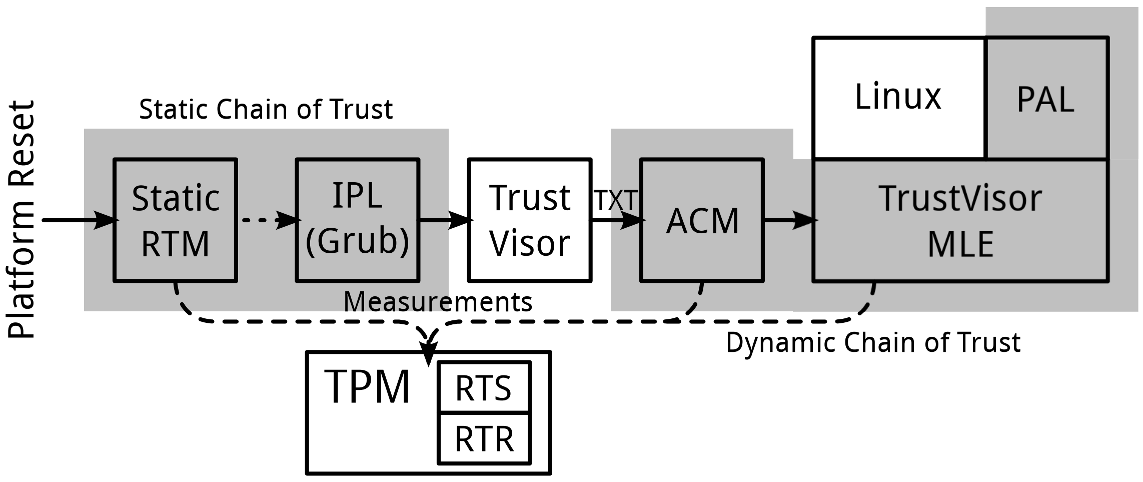

FIGURE 4.3: Diagram showing the execution of TrustVisor with Linux as guest operating system and one PAL as part of the dynamic chain of trust.

4.3 TrustVisor: a Hypervisor for Minimizing Application's TCB

TrustVisor [MLQ+10] aims for the same target as Flicker, it tries to minimize the TCB for small applications (PALs). But instead of doing the measured launch every time a PAL shall be executed, it is only done once for TrustVisor itself. This then will protect itself by using the hardware virtualization techniques introduced in 2.2 on page 7, making it a hypervisor as well.

This process begins at the same point at which TBoot starts, right after the boot loader (as can be seen in Figure 4.3). It also behaves largely like TBoot in this phase. The IPL loads TrustVisor, which will initially bootstrap itself and setup the dynamic chain of trust. It will then do the measured launch and start the main TrustVisor application — resulting in measurements being taking into the system's TPM.

At this point TrustVisor and TBoot diverge. While TBoot will yield control over the system to Linux, TrustVisor will setup itself as VMM, controlling the root-operation mode of the system and its IOMMU. With these techniques, it will only protect itself and the later registered PALs, all other operations and control of the system are given to an (unmodified) regular operations system like Linux or Windows. These will operate like they would normally do, with the difference that TrustVisor defines special traps (*hypercalls*) that make it possible for normal applications to interact with the VMM. Their main purpose is to start PALs and to receive results after they have finished.

During the creation of a PAL on TrustVisor, the VMM will unmap all physical addresses belonging to the PAL from the *extended* and *IOMMU page-tables* of the guest operating system, protecting them from any unauthorized access. After that, the PAL can be invoked and the VMM will let it execute, separated through the hardware protection from the untrusted operating system and any DMA device.

Because of this, McCune et al. argue [MLQ$^+$10] that as long as the user trusts the hardware and the VMM to uphold this separation, then the operating system is not part of the PAL's TCB anymore, reducing it to the hardware, the VMM and the PAL itself. This still holds true if more than one PAL is created on the system; it is the VMM's responsibility to separate these from each other, too.

Unlike with Flicker, this process doesn't require repetitive use of the measured launch, resulting in a better performance. But also unlike Flicker, it is not possible to start TrustVisor during the runtime of a normal operating system, it may only be started by the IPL. And lastly, another shortcoming it shares with Flicker: it currently supports only 32 Bit operating systems as guests.

4.4 Other Works

Next to these works, mainly on the issue of making use of the new dynamic chains of trust, there is also constant work on other issues surrounding the topic of trusted computing. Parno et al. give a very comprehensive overview over this field in their work “Bootstrapping Trust in Modern Computers” [PMP11].

One particular issue, that also concerns this work, is how to gain knowledge of the TPM's public keys and take ownership over it. This process was described it Chapter 3 on page 25. To gain trust in the system in question, it is necessary to provide convincing evidence, and one part of this evidence is to prove that the provided measurements are taken on the same system and are stored securely in the TPM. This evidence is given by using the *Quote* mechanism, which signs the measurements with a special key (an *AIK*), of which the public part is *known* to the verifying party and the private part is only de- and encrypted in the TPM to which it belongs.

The security critical question in this process is: does the known public part of the signing AIK really belong to the TPM in question? One possible answer could be the TCG's certification system that roots in the certificate for the TPM's *Endorsement*

Key, certifying that the EK belongs to a trustworthy TPM. If this *would* exist, then it could be used to certify, for example, that an AIK belongs to the TPM and thus is only used to sign measurements stored in it. But in practice this system never caught on [PMP11] (for example, the test system available for this work didn't contain such a certificate)

Because of that, the only remaining way is to retrieve the public AIK at the time it is created and to trust its authenticity as a fact. The same problem applies to the process of taking ownership over the TPM: at the time this is done, there can not be any evidence that the system isn't compromised already, because the TPM is not yet fully functional (a chicken-and-egg problem). As of now, it was not possible to find a satisfying solution for both of these problems.

5 Design of the Trusted Hypervisor Execution

Up until now, this work has shown what the new virtualization techniques on x86 can accomplish: it is now possible to implement virtual machine monitors that can perform processor and I/O virtualization with minimal emulation efforts, solely through the use of hardware. This reduces the amount of necessary source code for such a VMM drastically.

But, with the example of Jailhouse (see 2.3 on page 15), it was also shown how using these techniques can raise the importance of a single piece of software, up to be the single point of failure for a potential high number of depending guests. Therefore it was shown in Chapter 3 on page 25, that in order to *trust* this system, there has to be a way to provide convincing evidence that the correct hypervisor was executed and has not yet exited — with the requirement that the hypervisor can protect itself sufficiently during its runtime. The method chosen in this work to make this process work — Intel TXT — was then introduced in 3.4 on page 40, along with the process involved to create an environment to securely measure and store hashes about the loaded piece of software. Because TXT does not require a continues process of measurements — from the beginning of the boot process, up to the point where the hypervisor is launched — the resulting TCB will not have to contain as many components, but will only consist of the involved hardware, the ACM and the launched software.

Combining this gained knowledge, this chapter will present a design for a hypervisor that can be started with Intel TXT, during the runtime of a already running operating system. After the measured launch, it will then bootstrap the system again and resume the previously executed operating system as one of its guests. The main example for such a hypervisor in this work is Jailhouse, and the design will thus be fitted for its requirements and already existing architecture (see 2.3 on page 15).

Starting in 5.1, an overview over the solution created in this work will be shown, including all general steps and components it has to make and makes use of. After that, in 5.2 on page 64, it will be defined what the MLE for Jailhouse consists of and how it is structured. In the last three Sections 5.3 on page 70, 5.4 on page 72 and 5.5 on page 73, the responsibilities of the main components in the design will be specified.

5.1 Design Overview

The general overview over the designed process to start Jailhouse with the help of Intel TXT can be seen in Figure 5.1 on the facing page. All components of this figure that are shaded in gray have been changed or were added to Jailhouse in this work to make the measured launch possible (a more in depth description of each will be given in the sections to follow).

First of all, **the MLE has to be assembled**, and it has to be decided what the contents of this environment shall be, allowing for the requirements raised by TXT and Jailhouse to be fulfilled. The result here consists of a TXT stub — the piece of code where the ACM will jump to, logically separated from Jailhouse —, the Jailhouse image, a gap for its per-CPU areas and its complete configuration.

The main reason for extending the Jailhouse image with the **TXT Stub**, rather than integrating this code into the hypervisor code itself, will be given in Section 5.2 on page 64. But apart from this, it also makes it possible to reduce the necessary changes required in the main Jailhouse hypervisor code — reducing build dependencies and the possibility to introduce bugs that would also show when executed without TXT support.

The Jailhouse loader is extended in a way, so it can first **load this MLE**, just as it would normally load Jailhouse, and then give control to an additionally added **TXT loader** (only when a TXT enabled image was loaded). Subsequently, this will then take all necessary steps dictated by TXT to **load and configure** the ACM (this may optionally include the load of LCP data). The ACM in turn will later measure the loaded MLE. The last step of the TXT loader is to **start** the measured launch by executing the SMX instruction **GETSEC**.

At the *end* of the measured launch, the ACM will yield control and **jump into** the MLE's TXT stub. After the system's environment was noticeably changed during

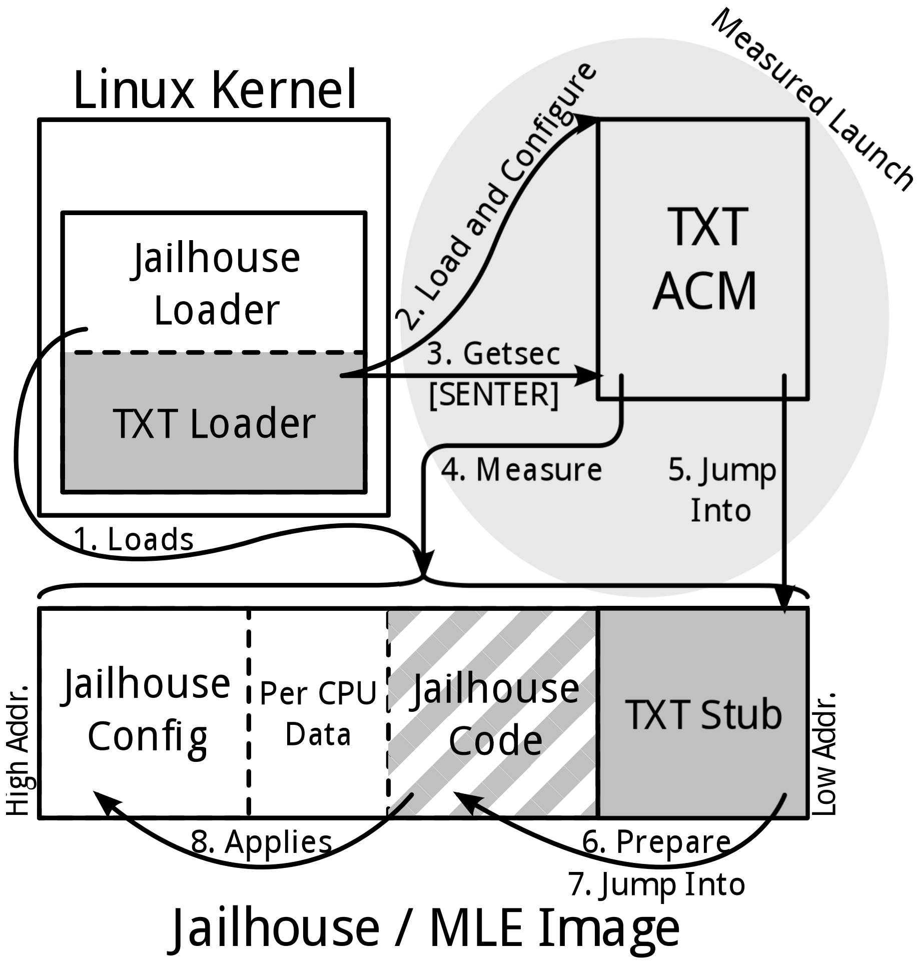

FIGURE 5.1: Diagram showing the general overview over the process of starting Jailhouse with support of Intel TXT. This can be compared to the diagram previously shown in Figure 2.7 on page 20. The areas shaded in gray are the areas of this work (except the measured launch), they have been either added or altered in order to make the measured launch possible.

the launch, it is now the responsibility of the TXT stub to bootstrap it again and bring it back into a suitable state for Jailhouse to execute. The final result after this phase is either a TXT reset — the error path —, or an other **jump**, this time **into** the nearly unmodified **Jailhouse code**.

Jailhouse will behave as it would without the measured launch, with the exception of a few necessary changes. It will **apply** the measured configuration and start executing the root cell, with the states of the operating system that was running before the launch as part of the guests-states in the VMCS.

The overall outcome can be compared to the one accomplished with TBoot (see 4.1 on page 53). The dynamic PCRs of the system's TPM will be reset during the SMX operations, and they will be extended by the used ACM's measurement in

PCR 17. The ACM in turn will extend the measurement of the MLE into either PCR 17 or PCR 18, depending on a configuration option set during the TXT loader phase. Trust can then be established by using any Linux TPM software stack that supports the **Quote** operation and any AIK that is bound to the system in question and known to the trusting party (see 3.3.1 on page 36) — there is no need to re-implement this step inside the MLE, the measurements are secured in the TPM.

Should Jailhouse ever shut down, it has to finalize its execution by running the appropriate SMX instruction and by extending at least one additional value into PCR 17. This has to be done to signal, upon a following remote attestation, that Jailhouse is not running anymore (compare with 4.2 on page 55)[1].

5.2 Defining the Parts of the Hypervisor's MLE

As first part of the design, the Hypervisor's MLE has to be defined. On the target system, this will define what is measured by TXT's SMX and ACM during the measured launch. Anything not included here is not measured and hence can not be used without comprehensive checks for its validity.

The following components are required to be included in Jailhouse's MLE:

1. the MLE needs to include the *Jailhouse image*;
2. it needs to contain all configuration parameters for Jailhouse to execute, it needs to include the *Jailhouse configuration*;
3. it needs to contain code suitable to execute after the ACM, especially after this has set the common processor state before the jump (see 3.4.2 on page 47).

To trust a hypervisor, the system has to measure all components contributing to the function of the VMM in control during the root-operations, before it gains this control (this specific point of the measurement was discussed in 3.2 on page 28). The process of how Jailhouse is started and configured was explained in detail in 2.3.1 on page 19. Both components, the image and the configuration, are integral part of this process and will determine the function of the resulting system. Moreover, they will also determine which parts of the system's hardware will be controlled by

[1] This nal step is not yet implemented in the implementation created during this work. Instead, it does always reset the system completely in case Jailhouse is stopped, and thus also resets the PCR values to prevent forging. Anything like this would require at least parts of a TPM software stack.

Jailhouse and with what security settings. They need to be measured and hence are to be included into the MLE.

To still allow Jailhouse to execute without hardware support for Intel TXT (as it also aims to support AMD and ARM), the image and configuration can not be changed in a way that would make the normal start of Jailhouse impossible. Rather than changing the old entry-path completely, there will be a new entry-path added for the use with TXT (the TXT stub, see below; there will still be some minor changes necessary to the old path).

Additionally, Jailhouse itself adds some more requirements that need to be fulfilled by the MLE:

- Jailhouse on x86 is always executed in a 64 bit environment, the code of the image is always compiled with *IA-32e* as target operating mode (also called *long mode* or *x86_64*);
- its address space starts at the fixed virtual address of `0xfffffffff0000000`, while the physical position is unknown at the compiler- and link-time;
- the configuration is located after the image, with a gap for Jailhouse's per-CPU areas.

The first two requirements can not be fulfilled immediately after the ACM jumps into the MLE at step 5 in Figure 5.1 on page 63 (more details follow in 5.2.1 on the next page). This makes it necessary to include code into the MLE that will prime the system for the use with Jailhouse — the corresponding part in the previously referenced image is the *TXT stub*. At the same time, this will solve the problem of adding a second entry-path to Jailhouse: the *TXT stub* will be able to prepare the environment in a suitable way, so it will be possible to run Jailhouse itself nearly unchanged.

What Can Not be Part of the MLE Several other parts of Jailhouse and its normal execution environment can not be part of the MLE though:

- The *Linux kernel module* that Jailhouse uses to load and start its image can not be included into the MLE. It is part of the Linux kernel and uses several functions of it in order to work. Those would all have to be included into the measurement as well — and with that into the TCB. By including their required functions as well, it would likely required to include the whole kernel into the TCB and thus defeat the whole purpose of using TXT in the first place.

- Any data passed from the module to Jailhouse which are not already known at the time where the image and configuration are created. This primarily means the *processor state* of Linux and the Linux *page table* that is used until Jailhouse replaces it with its own.

In case any part of the MLE makes use these information, however they are passed, it has to evaluated them thoroughly for any wrong or possibly malicious parts.

5.2.1 Structure of the TXT Stub

As seen, for most parts the MLE will be unchanged from what is currently part of the Jailhouse execution environment set up by its kernel driver. The biggest change is the addition of the *TXT stub* at its beginning.

One reason to separate this part from the rest of the MLE is to minimize the changes necessary to the existing Jailhouse code base. By doing so, it is possible to construct the remaining changes in Jailhouse in a way completely neutral to TXT (see 6.5 on page 92). But next to this, there is also a hard technical argument, arising from the requirements set by Jailhouse and the ACM.

Changed Processor State After the Measured Launch

During the description of the measured launch in 3.4.2 on page 47, it was stated that before the ACM will yield control to the MLE, it will set a common processor state (see Table 3.9 on page 48). This state can not be influenced from the outside and thus: every code immediately run after this point has to confirm to this state.

In this environment, the MLE has unrestricted (read-only-) access to the full 32 bit address space. Every used virtual address is translated into the same linear and the same physical address (the code segment starts at 0 and allows for accesses till 4 GB, and there is no paging enabled).

This means that the normal hypervisor code can not be the target for this jump. First of all, it is compiled using the Long Mode as target, and secondly, it expects to be mapped to the virtual address space above `0xffffffff0000000` (see 2.3.1 on page 19) — no matter were it is placed physically in the main memory. Both requirements can not be fulfilled by the initial MLE environment. It first has to be primed for the hypervisor by another piece of code.

This is also a major difference between this solution and the ones previously done (compare to Chapter 4 on page 53). TBoot and TrustVisor are both started by the IPL, they are both compiled with 32 bit as target and they have a fixed address space whose start they can specify via the Multiboot protocol. Flicker on the other hand also has the address space problem, but also requires a 32 bit platform to work. This makes it possible for all three previous solutions to be direct target of the ACM.

Every MLE that is not designed and build with the TXT target mode as target for itself has to add a separate component that makes a transition into the expected mode possible! In the case of Jailhouse, these parts can also not be build in the same process, with the same binary as target. The used linker (`ld` from the GCC) can not link two objects with different used address lengths[2] [GNU13] (will be shown in 6.2 on page 78).

Specifying the Stubs Address Space

The change of the processor state has also another implication in the design: the stub can not use the same address space as Jailhouse.

In contrary to TBoot and TrustVisor, Jailhouse can not be programmed with a fixed *physical* start address that is already known at the compile time of the image. This means, the address can also not be hard coded into the address layout applied by the linker (or calculated by it). The physical starting point is only just stated in the Jailhouse configuration and thus only known at the runtime of the compiled image. Additionally, the same requirement as in the argument for the previous decision applies: the fixed virtual addresses of Jailhouse starts *above* 4 GB.

Both requirements are in conflict with the changed processor state. After this state is applied, there is no address translation activated, every reference will translate from virtual into the same linear and the same physical address, which are also all at most 32 bit wide. Together with the unknown physical position, this means that the Stub can not share the static address space of Jailhouse.

During this work, two possible solutions for this have been found: reusing the MLE page table, or applying a position independent address scheme to the stub.

[2] To the best of our knowledge, there is no such linker.

Reusing the MLE Page Table One of the two ways found is to reuse the MLE page table created for the measurement done by the ACM. During the setup, a pointer to the beginning of this page table is stored on the TXT heap — a centralised data structure that TXT uses to communicate data between the different parts of the dynamic chain of trust (for more details see 6.1 on page 77); depending on the ACM version, this information may also be placed in the ECX register (see Table 3.9 on page 48). Because the exact location of this TXT heap can be found via a TXT chipset registers, it is possible to retrieve this pointer again once the TXT stub is in control over the system.

Once the MLE has retrieved this pointer from the heap, it can use it to re-enable the paging of the system (it will still be running in 32 Bit protected mode, but with paging enabled). It thus can use its own private address space with absolute addresses — like every normal user application —, it only has to follow the general restrictions imposed by the MLE page table (no gaps in the address space, 32 bit addresses).

The downside of this is, although this page table is used to measure the MLE and is used to resolve the entry point, the exact mappings between physical addresses and virtual addresses are not measured. There is no certain way for the MLE, other than reverifying the table, to tell what physical addresses are actually used — in case absolute physical are use in the MLE (e.g.: MMIO), this is a potential security.

Applying a Position Independent Address Scheme The other possible found way is to apply a technique known from dynamic linking of libraries into binaries at runtime: *PIC* [Ben11] (*Position Independent Code*).

As can be seen in Table 3.9 on page 48, the ACM does not only jump into the MLE, but it also puts the target address into the register `EBX`. This means, during the execution of this first address in the MLE, it knows about its physical location and it can use this information to calculate every other address relatively to this one.

To make those calculations easier, it is possible to design the MLE in a way that the entry point is always on the first page of the mapped space (within the first 4 KB of its address space) and that the start address of the MLE also falls onto the start of this exact page. The stub can then align the given address of the entry point to the nearest page boundary and thus has derived the beginning of its own physical address space.

As an example: lets assume the virtual address space of the MLE begins at the address $a = 0$ and its entry point is at address $e = 768$. By instructing the linker with the start address a, it will calculate every final address of the binary as offset from 0. This address a is also stated as the (linear) start address in the MLE header. The system is therefore also required to use a page boundary as physical start address.

Now, lets assume further that the physical start address, that corresponds to a, is decided to be p = `00003000h`. This means that the physical entry address, that corresponds to e, must be i = `00003000h` + e = `00003300h`. This is the value stored in the registers `EIP` and `EBX`. Upon getting the control, the MLE can now calculate the beginning of its address space by doing a simple and-operation: $base = i$ & `FFFFF000h`. And because all virtual addresses, created during the linking-step of the stub, were created as offset from $a = 0$, their physical addresses can be calculated during the runtime by simply adding them to the calculated $base$.

When applied in this manner, this kind of *PIC* can even be supported by compilers like the GCC [Ben11], making it possible to compile unchanged C code for it, with only a very short initialization necessary (see later in 6.4 on page 86).

The Resulting MLE for Jailhouse

For the example used in this work, the hypervisor Jailhouse, no significant advantages or disadvantages for either approach could be found. Both will require initialisation and verification in assembly (for the former the argument was made in the same paragraph, for the later it will be made in 6.4 on page 86), but once this is done, both support the implementation of the main code base in unaltered C with native support by the GCC C compiler.

The final decision in the design for this work was made in favour for the PIC approach. Because Jailhouse makes it necessary to create yet an other page table, for the change from protected mode to the IA-32e mode, it also requires the implementation of a paging scheme in the stub — different from the PAE paging used for the MLE page table. To additionally support the use of the MLE page table, it would be necessary to either make this algorithm more complex, to allow for a conversion from PAE to IA-32e page table, or alternatively, to implement two independent page algorithms — both enlarging the TCB.

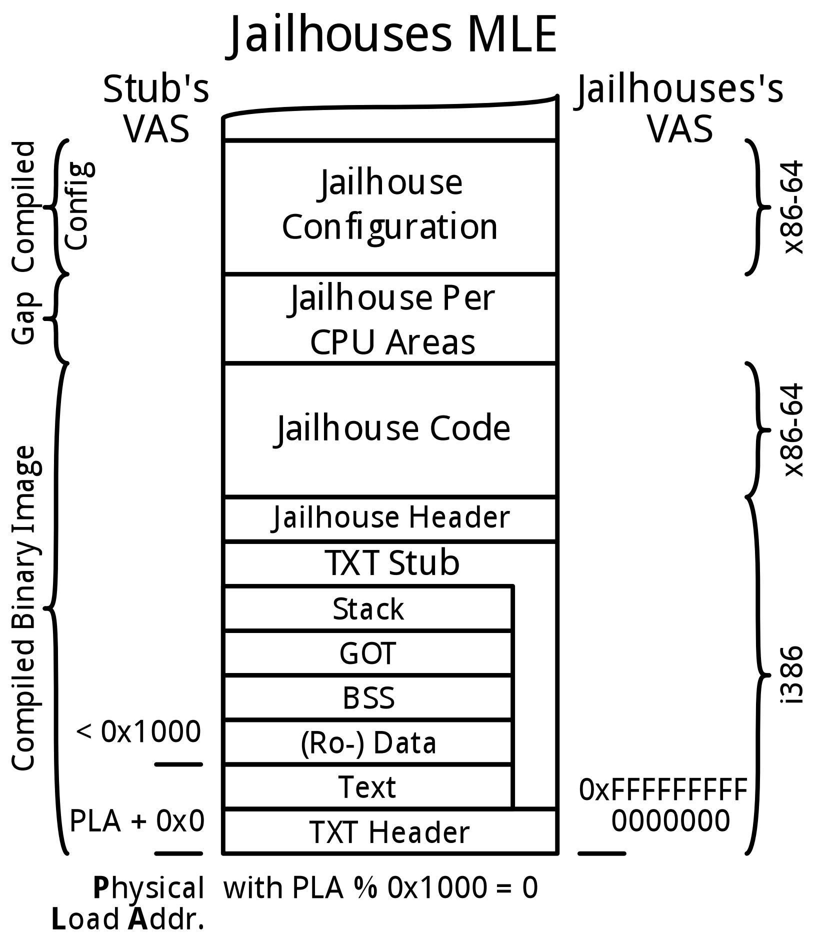

FIGURE 5.2: Overview over the resulting MLE layout for the Jailhouse hypervisor. The *VAS* is the *Virtual Address Space* as used during the execution of the corresponding Jailhouse part. The *Gap* is lled with zeros during the time of measurement, but is required by Jailhouse's Memory layout.

With this and the decision to split the hypervisor into two separate parts, the MLE will finally result in the layout displayed in Figure 5.2.

This layout, including everything up to the end of the configuration, will be part of the measurements taken during the measured launch by the ACM. It represents the core of Jailhouse's TCB. The only other components are the ACM and the hardware itself.

5.3 Responsibilities of the TXT Loader

In the overview shown in Figure 5.1 on page 63, the first new component added to the Jailhouse hypervisor is the *TXT loader*. The task of this component is to setup all the required configurations to satisfy the requirements set by the TXT specification, in order to subsequently start the measured launch.

Later, during the description of the implementation (see 6.3 on page 80), it will be shown in detail what steps exactly are necessary for this to work. This will also show that most of these steps are done to setup the ACM and only a few for the SMX call that starts the measured launch. But because the specification of TXT is very explicit about these details, it is not required for the design to further narrow them down.

The main design decision about it was implicitly made by not including it into Jailhouse's MLE. It will be an extension to the already existing *Jailhouse loader*.

Because the loader also needs to be able to support hardware and images with and without support for Intel TXT, like the Jailhouse image in the MLE, it is necessary to separate the TXT part logically from the normal parts of the loader. But unlike the TXT stub, it will be possible to implement and build them within the same context, making the interaction between the two easier.

In combination with the Jailhouse loader, it will be responsible to:

- load the necessary data from the long time storage into their correct position according to the designed MLE layout;
- load optionally possible LCP data (those will be placed after the MLE, but are *not* part of it) — the part of the LCPs stored in the TPM will decide whether this is necessary or not;
- create the MLE page table and place it at the correct position according to the specification;
- configure the DMA protection for all the elements named before, so that they may not be changed *during* the measured launch (after they have been measured), TXT specifies the methods to be used;
- load the ACM from the long time storage into its specified position and configure all the necessary options for it;
- save the state of all processors in an appropriate place (as they are reset by SMX and the ACM);
- instruct the measured launch.

Details about this process will later be given in the implementation, at 6.3 on page 80.

5.4 Responsibilities of the TXT Stub

The second and last truly new component to Jailhouse is the TXT Stub. How it is integrated into the MLE and why was shown in 5.2.1 on page 66. The reason — to prime the environment according to the requirements set by Jailhouse — also implies the responsibilities it has in the design for the TXT support.

In contrary to the TXT loader, the stub is not specified in nearly as great of a detail by the TXT specification. Most of its responsibilities are dictated by the state of the processors after it received control from the ACM and Jailhouse's requirements:

- it needs to bootstrap the environment again, to be able to receive and process exception, to have a *writeable* data and stack segment again, and to be able to execute compiled C code;
- it has to create the page table for Jailhouse and to map all necessary memory locations according to the memory layout;
- all the other processors that were disabled during the SMX operations of the measured launch have to be re-activated (using another specific SMX instruction);
- the same has to be done to the external events that were disabled during the SMX operations (e.g.: SMIs, NMIs, IRQs);
- it has to prime some of Jailhouse's structures (see below);
- finish its execution by changing the operating mode to IA-32e and by jumping into the main Jailhouse code.

To not endanger the integrity of the MLE, all those steps have to be done without the use of external and unmeasured data. This especially holds true for the processor states that were saved by the TXT loader — it is not part of the MLE and hence can not be trusted. Most information though can be retrieved from the also measured Jailhouse configuration (it describes the target system exhaustive; see 2.3.1 on page 19). All the dynamic runtime states, those that will be necessary to continue the execution of original Linux, can be securely handled during the execution of the Jailhouse core by putting them into the VMCS's guest-state area — this will virtualize them, but not apply them directly.

Nonetheless, it will later be shown in the implementation that this requirement can not be complied to at all the time. Processor states that are changed, but can not be put into the VMM's VMCS, have to be restored otherwise to guarantee the guest's function (the example later shown concerns the processors MTRRs).

In the last step before jumping to Jailhouse, the stub will pre-initialize Jailhouse's per-CPU areas and its paging structures. Jailhouse would normally save all the processor's states itself and it would calculate its own paging structures as well — while still using Linux's page table. But the former is not possible in the MLE, as it would require the re-set of the stored and untrusted states, and the later is redundant, because a new page table is already created at this point in the stub.

5.5 Changes in Jailhouse

Design and overall function of the main Jailhouse code have not been changed in this design, as it was required. Concerning the trusted execution, this especially means, it has to protect itself suitable from any external influences. The only notable changes had to be done to the initialization after the TXT stub and to the shutdown.

The start procedure had to be extended by a second path. On this new path, it has to take into account that the TXT stub has already saved all the processor's states, into the appropriate places in Jailhouse's memory, and that it doesn't need to create a new paging structure.

At the time of an eventual shutdown, Jailhouse has to recognize that is was started by TXT. In case it recognizes this, it has to extend an other measurement into the TPM's PCR 17 and possibly also 18 (those two contain the measurements of the ACM and the MLE, see 3.4.2 on page 42), in order to prevent following software from imposing as the trusted hypervisor. And as second and last action during the shutdown, it has to instruct SMX to exit.

Changes to enable the TPM's remote attestation process are not necessary in Jailhouse itself (see 3.3.1 on page 36). Because the used AIK and PCRs are protected by the TPM, it is possible to use any existing software stack on Linux to solve this problem (this also spares Jailhouse the need to implement its own communication means for this).

6 The TXT Implementation for Jailhouse

After presenting the TXT support for Jailhouse in Chapter 5 on page 61, it is now necessary to show what steps are required to implement this design and in what amount of source code and runtime growth this results.

As one of the main motivations for Jailhouse is to have a small and verifiable code base that provides all the safety relevant features for the guests on its own, especially the growth of the source code is important for it. Although TXT contributes to its proper function by adding provable certainty that the correct hypervisor is used on a target system, it should not be necessary to enlarge the code base for this by an unreasonably amount of new sources.

To quantify exactly this and to show that the presented design is feasible, it was implemented during this work. This chapter will present the main steps that were necessary for this. Furthermore, it will name all restrictions that the current state of the implementation has and how they are planed to be fixed in the future.

The first Section 6.1 on the following page of this chapter will present the basic programming structures of Intel TXT. Following that, Section 6.2 on page 78 will give a description of how the new hypervisor image is build according to the design. Sections 6.3 on page 80, 6.4 on page 86 and 6.5 on page 92 will then given a detailed description of the main steps necessary to implement the new components introduced to the Jailhouse hypervisor, and Section 6.6 on page 93 will afterwards discuss the remaining issues of the implementation of those steps. Finishing this chapter, Section 6.7 on page 95 and 6.8 on page 97 will present the relevant size and performance metrics of the current implementation.

6.1 Programming of the TXT Components of the System

Next to the added SMX instruction set, TXT adds two more central structures which are necessary to implement the support for it: the *TXT configuration space* and the *TXT heap*. Both are created by the system, the config space by the chipset and the heap by the firmware at boot time.

The TXT Configuration Space

The config space is a set of registers located on the chipset. They are programmed via a memory mapping which simultaneously exists at the addresses `0xFED20000` and `0xFED30000`. Each of these mappings is one 4 KB page long and contains only register with a width of 64 bit.

They both represent a different view on the same space (they both have the same set of registers at the same offset). The mapping that starts at the address `0xFED30000` is the always accessible *public* view. It mostly provides read-only access to the mapped registers (with a few non-security relevant exceptions). The mapping at the address `0xFED20000` on the other hand is treated as the *private* view. It is only accessible in between the successful execution of the instructions `GETSEC[SENTER]` and `GETSEC[SEXIT]` (during the dynamic chain). Outside of these, every read will return zeros and writes will be ignored. The access to the registers is handled with the normal set of MMIO operations (depending on the paging settings, this space has to be mapped into the virtual address space and set to uncachable)

The complete set of available registers can be found in the TXT specification, Table 6.1 on the facing page lists to most important for this work.

As shown, these registers also contain the location of the TXT heap, and they specify the memory where the setup has to put the ACM before it can start the measured launch. Both areas are created and locked by the system's firmware (e.g. the BIOS).

Additionally to this, the space which holds those two areas is also called *DMA Protected Range* (abbr. *DPR*) and it is protected by the chipset from every DMA access on the system. This protection is applied after the IOMMU applies its own mappings and is also activated if the IOMMU is not, even outside of any SMX operation [Int14c].

Register Name	Description
Status	Shows information about the state of TXT in the system (was `SENTER` performed, what locality is opened by the chipset, is the private space open, etc.)
Error code	The only information that is saved during a *TXT reset* (a system reset caused by an error during the measured launch).
Reset	A write to this will induce a TXT reset from within the MLE.
SINIT Base and Size	Space reserved by the system's firmware to put the ACM at runtime.
HEAP Base and Size	Space reserved by the system's firmware for the TXT heap.
Open and close locality 1 and 2	Can be used after the measured launch to open and close the use of locality 1 and 2 via the chipset (3 and 4 remain off limit).
Public Key	Stores the public key used to verify the ACM during the measured launch.

FIGURE 6.1: Table showing a selection of registers mapped in the TXT con guration space.

The TXT Heap

The TXT heap is the memory space used to communicate between the different components of the measured launch — it is not memory mapped, but normal space of the system's RAM. Like said, it is created by the firmware and discovered through the TXT config space.

The exact structure of the heap is defined in the TXT specification and is divided into 4 main blocks:

1. **BIOS Data**: contains data originating in the system's BIOS. The main purpose of this block is to communicate the location of the ACM, in case it is provided by the BIOS and not Intel. The same may be true for any LCP data provided along with this BIOS ACM. Both could not be seen on the test hardware during this work.

2. **OS to SINIT**: this contains the configuration of the ACM that is done by the setup software. It will point to the root of the *MLE page table*, the *MLE header* and the MLE size. Furthermore, it will contain the location of any

LCP data, in the case the system uses this feature (see 3.4.3 on page 48), and it will contain other minor configuration details.

3. **OS to MLE**: this is the location for any data that the setup software wants to provide to the MLE. It is not defined any further and in the case of Jailhouse, it will be used to save the processor state of Linux before the measured launch.
4. **SINIT to MLE**: this block will contain data computed by the ACM during the measured launch and it can be used by the MLE to learn trusted information about the system (hash values of different components, memory mappings and more). The Jailhouse MLE doesn't rely on these information, but it rather uses the comprehensive and also measured Jailhouse configuration.

It is not exactly specified how large the reserved space for the whole heap is. On the test hardware in this work it was 768 KB big. This means, the setup has to be careful about what data it puts into the "OS to MLE" space, in order to leave enough space for the ACM to store its own data in the last block of the heap.

6.2 Build of the Hypervisor Image

The main parts of Jailhouse's MLE are the Jailhouse image and the TXT stub — next to the configuration, which is not changed. Why those two parts are separated from each other has been shown in the design, at 5.2 on page 64. In 5.2.1 on page 69 it was further shown how the address layout of the two is defined. But because those two parts and their layout don't confirm to the defaults used by the toolchain, it was necessary in the implementation to figure out a way to make this build possible nonetheless.

As the main toolchain for compilation and linking, Jailhouse utilizes the GNU Compiler Collection and the GNU Binutils. The GCC is able to compile C code with the desired target architectures by setting a fitting CPU type as argument to its command line option `-march=<target>` (by setting this to `i386` for the TXT stub, the GCC is prevented from using any unsupported or not yet activated instructions). Additionally, the length of the used addresses and types can be chosen by either using `-m32` for 32 bit and `-m64` for 64 bit code.

But because the two parts are build with different target architectures, it is not possible to link the resulting object files together into one final binary. The code of the TXT stub uses 32 bit addresses and Jailhouse uses 64 bit addresses, those

```
SECTIONS
{
    . = 0;
    __mle_start = .;
    .header     : { *(.header) }

    . = ALIGN(16);
    .text       : { *(.text .text.*get_pc_thunk.bx) }

    ....

    . = ALIGN(16);
    .stack      : { *(.stack) }
    __txt_stub_end = .;
}
```

LISTING 6.1: Code listing showing a snippet from the TXT stub's linker script.

can not be mixed by the linker. This is solved by compiling and linking both parts separately into their own binary. Those are then stripped from their ELF headers with GNU `objcopy` (the headers are not necessary for Jailhouse) and the outcome is glued together into one final image (in the presented address layout, this is the part from the beginning TXT header, up until the end of the Jailhouse code).

The information where the TXT stub ends and the Jailhouse image starts is stored in the TXT header. This is for example used in the Jailhouse kernel module during the setup of the measured launch. The code of the TXT stub on the other hand doesn't need this information explicitly, it can make use of the virtual address layout instead. During the final linking step, the linker will know how long the stub will be — to calculate the final addresses of all defined symbols — and the code of the stub can use this information by defining a symbol that is right at the end of it.

This can be done with linker scripts supported by the GNU linker `ld` [GNU13]. Those scripts are also used to define how exactly the virtual address space of the stub shall look like — to make sure it confirms exactly to the one described in the MLE layout, and to define extra areas needed during the runtime of the stub. The relevant part from the TXT stub's layout is shown in Listing 6.1.

In this listing, it can be seen that the beginning of the layout is specified at address zero — to make the PIC addressing easier (see 5.2.1 on page 68) — and that it ends with the definition of the symbol `__txt_stub_end`. In between, it collects all the code generated by the compiler in the different sections, like for example the `text` section for source code or the (not shown) `data` section for pre-initialized variables. Symbols like `__txt_stub_end` can then be used in the source code as addresses and will be filled during the final linking step.

Putting all this together with GNU `make`, it will build both parts and merge them into the final hypervisor image.

6.3 Implementation of the TXT Loader

The responsibilities of the TXT loader have been shown in the design at 5.3 on page 70. This section will show the main steps that are necessary to fulfill those.

The loader is invoked by the already existing Jailhouse kernel module. This follows the same operations as it would without TXT: it loads the hypervisor image (now with the TXT stub pre-attached), interleaves the area for the per-CPU structures and loads the Jailhouse configuration in the following space. The *physical* space used for this is defined in the Jailhouse configuration and reserved for this purpose via a kernel command line parameter — the virtual space is reserved as shown in the address layout of the MLE (see 5.2.1 on page 69). Following this load, the module detects the presence of the TXT header — instead of the usual Jailhouse header — and invokes the TXT loader.

During the runtime of the loader itself, the main task is to configure the *ACM*, not the SMX instruction — `GETSEC[SENTER]` in case of the loader. This instruction only takes 4 parameters: the selection of the operation `SENTER` in the `EAX` register, the physical position and size of the ACM in the `EBX` and `ECX` registers, and optionally a selection of to be disabled SMX functions in the `EDX` register (from the functions available in the current SMX implementation, none can be disabled without breaking the measured launch, leading this to be be always zero). Apart from these parameters, the only other two conditions are that the instruction is run in the highest privilege level, outside of any VMX operations, and that the ACM was loaded into the DMA Protected Range discovered via the TXT configuration space (see 3.4.2 on page 42 for more details about how SMX handles to ACM).

But all this does not yet specify anything about the MLE. The handling and measurement of the MLE is the task of the ACM in Intel's dynamic chain of trust[1]. This section will now discuss the necessary steps to configure the ACM in a specification conform manner to measure Jailhouse's MLE.

[1] As opposed to AMD, where the concept of an ACM doesn't exist at all [AMD13].

Placement of the Loaded Hypervisor Image The exact location of the hypervisor image, at the time it is loaded in the Jailhouse kernel module, is influenced by two constraints required for TXT: the physical position has to be in the lower 4 GB of the memory, and it has to be placed *after* the MLE page table — the former has to be considered at the time the Jailhouse configuration is created.

Calculating the space required for the MLE page table can be done without creating the table itself: the size of the MLE is known at the time the image and configuration is loaded, and the virtual start address of the MLE is a constant specified in the address layout (zero).

The result of this step is a fixed amount of blank space at the beginning of the predefined Jailhouse memory — enough to locate the MLE page table — followed by the MLE.

TXT Pre-Checks Before the loader can start the setup itself, it has to check whether the processor and system support the measured launch in the first place.

It does this by first checking whether the processor supports SMX (via the instruction `cpuid`), and in case it does, the loader activate SMX by setting the corresponding bit in the `CR0` register. It can then check for the necessary TXT functions via the SMX subcommands `CAPABILITIES` and `PARAMETERS`. This process is not much different from the discovery of other optional features in x86 (like the VMX extensions).

Some information learned this way have to be saved for future configuration steps: the minimal version of ACM and the type of memory caching supported during the measured launch (more details in 6.3 on page 85).

Load of Any Available LCP Data Whether or not the loader needs to load any LCP data can be found out by examining the LCP part that is stored in the TPM's NV RAM — this part is always required (see 3.4.3 on page 48).

In case the data part is required, then it also has to be protected from DMA, along with the MLE page table and the MLE (see next paragraph), but without any special placement required. The loader will place it after the MLE. And because it is only required by the ACM, it can be overwritten after the measured launch without further checks.

Because there was no time to implement the necessary TPM interaction to read from the NV RAM, this step is done unconditionally in the current implementation.

Configuring the DMA Protection for the MLE It was already discussed how the ACM and the TXT Heap are protected from DMA during the measured launch (the later with the help of the system's DPR, and the ACM by loading it into the BSP's ACM area). The MLE though, along with the MLE page table and the optional LCP data, also need to be protected from DMA. Otherwise they might be changed by the hardware after they were measured by the ACM.

For this purpose it is specified with TXT, that they have to be placed into a *Protected Memory Range*. These *PMR*s are a feature of Intel's VT-d [Int13] (*PMR* also stands for *Protected Memory Register* in this context).

PMRs are not part of the normal remapping structures of the IOMMU and they can be used without them. They can however be overwritten by the remapping structures (unlike the DPRs). Programming them is straight forward: they are discovered via the capability register of the IOMMU, they are split into two sets, each set with two registers, and they have one global control register. The control register is used to switch the protection on and off and the two register sets are used to define the protected ranges (one register as base and one as limit). Both ranges can protect a region in the lower 4 GB of memory, but only one can protect space above that limit.

But while this is straight forward, it is also troublesome in case the system in question already makes use of them and/or the IOMMU in general.

For the TXT support in Jailhouse, both restrictions can be ignored at the moment. Linux doesn't make use of the PMRs at the time of this writing and Jailhouse forbids the use of the IOMMU remapping structures before it is launched (this is enforced by another kernel command line option prior the boot). Should one of these preconditions change, it would make it necessary to re-program the PMRs before the launch of TXT — migrating the protection applied by the previous PMR settings and the remapping structures.

For the loader this means, it can monopolize the PMRs to protect the MLE and the other components. During the measured launch, the ACM is specified to check upon those settings before it takes any measurements [Int14c].

Priming the TXT Heap As explained in the description of the heap, the loader and the following TXT stub don't make much use of the heap.

Concerning the size of the heap, the loader needs to check whether it is big enough to contain all the processor states that are saved later during the setup, or not.

Other than this, the heap is only used to convey information about the MLE to the ACM (in the "OS to SINIT" section). It has to supply the size and position of the MLE, the MLE page table and the LCP data, in case any is used. And it also has to duplicate the applied PMR settings here. All of these settings will be checked by the ACM to evaluate the MLE.

Finding a fitting ACM Each ACM supports only a limited amount of hardware platforms (as far as it could be found out, Intel releases a new ACM for each new platform). The loader can discover if a specific ACM fits the target platform by examining its header.

The following are amongst the possible factors it has to check for (see the TXT specification for a full list [Int14c]):

- does the ACM support the platform's chipset and CPU;
- do the chipset and CPU have the same production state as the ACM (debug or release version);
- does the ACM have compatible features with the MLE?

The first two items can be determined with tables in the ACM header and the use of the TXT configuration space in combination with different discovery features of the CPU (`CPUID` and certain MSRs). In those tables the ACM states which chipsets and CPUs it supports.

The last item depends on a capability bit field in the head of both the ACM and the MLE. It can be used to signal the support for a set of features of TXT, but in the current version of TXT, most of these have to be enabled by default (an exception would be, for example, the TPM version, which can be 1.2 or 2).

After a suitable ACM is found, it has to be loaded into the memory described for it by the TXT config space.

Creating the MLE Page Table The MLE page table has to be created in the space reserved for it in front of the loaded MLE. A detailed description of the page table can be found in 3.4.2 on page 46, the process of creating it follows the specification of the PAE table layout.

Catching All Available Processors All the previous steps could be run without synchronisation between the processors. The loader will only execute them on the processor which started the Jailhouse enable. But after those are done, the loader goes into the final phase before doing the measures launch itself.

This requires amongst others the change and save of processor states, and although SMX will catch all processors anyway, this has to be synchronised to not loose any information when SMX starts and overwrites the states. Otherwise, it might later not be possible to resume Linux in the exact state as it was before the measured launch, and with that break its function.

Both the catching and the synchronisation can be done with functions provided by the Linux kernel (this will also disable interrupts and preemption). Important for TXT is to distinguishes between the BSP and the RLPs. Like explained in 3.4.2 on page 42, the actual launch can only be done by the BSP (recognized through the MSR `IA32_APIC_BASE`).

After this step is done, no other user or kernel process runs on the system, all processors are in the loader's code.

The first action at this point is to redo parts of the TXT pre-checks. Previously, the loader only enabled SMX in the `CR0` register of the initial processor. This value though is not synchronised across all processors, and thus has to be re-applied on all other processors as well.

Making Sure the TPM is Unused It might be possible that the TPM was in use before the loader caught all processors, and because it is only usable by one driver at the same time, the loader has to terminate the previous session in a clean way — making sure it is not activated at the time of the launch.

The loader does this by writing to the TPM access register of locality 0 and requesting it to become inactive. This single step can take up to 750ms [TCG05], but it has only to be done on one of the processors. After this time, the TPM is required to be in an inactive state.

Mapping the ACM in the Systems MTRRs While SMX sets up the system for the measurement of the ACM and during the subsequent run of the same, it will disable the paging of the processor. In this mode the processor has no access to the page level control of its cache (e.g., page table and PAT). Instead, it will use the *Memory Type Range Registers* (abbr. *MTRR*, [Int14b]) of the Intel architecture (unless caching is completely disabled with the `CD` flag in the control register `CR0`).

When activated, they apply a default caching type to every memory region not explicitly mapped, and otherwise they offer a range of registers (MSRs) to state the caching type for other specific memory ranges.

For the ACM, the loader has to apply a specific scheme: the space that is used by the ACM has to be mapped with the caching type *Writeback*, every other range of the memory (the default value, also including the MLE) has to be mapped with one of the types the loader found during the TXT pre-checks — the fall back is to map them with *Uncacheable*. This mapping has to be synchronised across all processors (MSRs are not synchronised by the hardware). It guarantees that the ACM can still profit from the processor's cache, but doesn't suffer from any unfit cache setting applied to the other areas of the memory — it will be predictable.

The old settings of the MTRRs, as they have been programmed by Linux, will be saved on the TXT heap (in the "OS to MLE" area), so they can be re-applied later in the MLE.

Saving the CPU State and Making the Launch After all those steps are done, the module has satisfied every precondition necessary to start the measured launch. Up to this point, an error or missed requirement will only result in a rollback and the release of all resources acquired during this process. The operating system will return an error code and otherwise can resume the operations unchanged.

Now though, the loader will enter the final phase of the setup. It will again gather all processors. All but the BSP will go on and save their current architectural state on the TXT heap, in the "OS to MLE" area.

To be later able to distinguish between the different states — in contrast to the MTRR state, which is global —, this is done in the form of an associative array. The associated index for this array is the processor's initial APIC ID. This ID is fixed to the processor and can not be changed across the measured launch (for processors where the ID can be changed during the runtime, `CPUID[0x01]` will always return the initial APIC ID [Kuo12]). Additionally, to save heap space, this array is designed to only have as many fields as processors in the system. Later in Section 6.6.2 on page 95, it will be shown what restrictions apply to this in the current implementation.

Subsequently to this save, the processors will signal to the BSP that they have finished saving their state and then they will stop executing by instructing `HLT` (interrupts are still disabled here).

The BSP meanwhile will wait as long as it has not yet received the word from all processors but himself. It will then save his own state in the same associative array as the RLPs and finally execute `GETSEC[SENTER]`.

This ends the setup phase and starts the dynamic chain of trust by establishing the root of trust for measurement, and it follows the steps described in detail in 3.4.2 on page 42. During this phase, every error will lead to a TXT reset — an error code (32 bit wide) will be stored in the error register of the TXT configuration space. The exact reason for such a reset can then only be found out after the system has booted again.

6.4 Implementation of the TXT Stub

After the measured launch, the ACM will eventually jump into the measured MLE. The exact entry point is given in the TXT header at the beginning of it (also as *linear* address that will be resolved via the MLE page table). In case of Jailhouse, this entry point will be part of the TXT stub and the responsibilities of that stub have been given in 6.3 on page 80.

The state of the (only) active processor at this point is determined by SMX and the common processor state set by the ACM (for the later see 3.9 on page 48):

- maskable interrupts are conventionally disabled;
- SMIs and external NMIs are also still masked;
- all other processors are sleeping and can only be reactivated via a way provided by TXT (a description will follow);
- the MLE, ACM and TXT heap are protected from DMA by the configured PMRs and DPRs;

- the TPM's locality 2 is open (locality 1 can be opened via the TXT config space).

The stub is now responsible to manage and restored these states — it will follow the order given in the design for this task. Other changes will remain off limit though; for example, the A20M pin will remain being masked unless the dynamic chain is ended by executing `GETSEC[SEXIT]` and the TPM localities 4 and 3 will remain closed (closed by the processor and ACM).

Verifying the Environment The first step in the stub is specific for the PIC method that is used to design the virtual address space of the MLE (see 5.2.1 on page 68). Because the MLE can not trust the setup to have mapped the expected *physical* positions — the ACM places no restrictions on the physical addresses, only on the linear ones —, it is required in both presented methods to verify the mappings in the MLE page table. For the PIC method, this needs to be checked right at the start of the MLE. If the MLE page table is reused, it needs to be checked when new addresses are mapped into the table (for example, to prevent double mappings).

In the designed memory layout for the Jailhouse MLE (5.2 on page 70), it is specified that the MLE is placed *continuously* in the physical space (because the same requirement is already made by Jailhouse without TXT). Because it concerns the physical addresses, this needs to be checked after the entry.

Because the linear start address is included in the measurement (in the TXT header), it is guaranteed that the first page of a *correctly* measured MLE is placed correctly. In this case, it is impossible, for example, to place the linear start address "0" on a physical address that is not a page boundary. But the next physical page is not guaranteed to be placed in direct succession — an ill-intended setup might leave a gap in the physical space.

The stub has to make a check for this manually and only with code that is located on this first page — otherwise it might already be manipulated. It can do so by using the rules set for the MLE page table (checked by the ACM) and the design of the MLE address layout.

Both the TXT header and the text-segment of the stub are place on the first page and thus are valid in case of a *correct* measurement. The stub thus knowns its linear start address, its length (both can be retrieved from the header) and it knows its current physical location from the `EBX` register. Furthermore, it knows that the MLE page table is required to not contain any gaps in the linear mappings and is required to have strictly ascending physical addresses. By locating the physical address of the first page — using the start address — it can calculate at what physical address the last page should be. If it sees that the last mapping does not result in the expected physical address, it knows that the page table is wrong — the only possible reason is a gap in the physical space.

Both steps only require an one-way look up through the page table — which in turn is only a succession of redirected memory lookups. If it finds that the last mapping is at the correct physical location, ***length*** addresses away from the first one, it knows that the MLE was placed correctly.

Initialize the TXT Stub Environment After the first verification is done, the BSP has to setup the general execution environment. This process is largely similar to the boot process of a normal operation system, with only a few exceptions due to TXT.

First of all, the stub has to replace the global descriptor table (abbr. GDT) with its own and then load the respective selectors for the code and data segments (no write access up to this point). The segments used by the stub will continue to use the *basic flat model* that was already used at the entry-point — every segment will have the same limits and map the linear address space from 0 up to 4 GB uniformly.

After the data segments are set, the BSP can also enable its stack. As seen in the memory layout of the MLE, it reserves a separated area for this purpose within the bounds of the TXT stub. The other processors that are still sleeping will not use this same area, but will later be initialized to already use their own stack space reserved in the Jailhouse per-CPU area. Because the BSP first has to figure out where exactly this per-CPU area resides within the MLE, it can not do this just yet.

With the stack enabled, it is possible for the BSP to enable its own interrupt descriptor table (abbr. IDT), to catch and process exception properly. At this point in the MLE's execution, the same strategy as in the ACM is applied in case of an error: the stub will issue an TXT reset and give a custom error code via the TXT config space. Any other handling, like for example return to Linux, would first required a operating mode change; this option is more viable later, when the MLE has already entered the execution of Jailhouse itself.

Although at this point, the stub could also enable the handling of IRQs again, the stub will not reactivate them, and neither will it reactivate the external NMIs (those were disabled during `GETSEC`). Both will later be reactivated by Jailhouse when it enters the execution of its VMX guests (VMX allows for the VMM to specify whether an external interrupt will cause an VM exit and it will enable interrupts and NMIs depending on this selection [Int14b, Int14a]). The default Jailhouse behaviour is to let the guests process external IRQs without a VM exit and cause a VM exit in case of a NMI.

The last step, before it is possible for the stub to use code compiled from C, is to initialize the *global offset table* (abbr. *GOT*) [Ben11]. This table is created by the C compiler when it is instructed to compile position independent code (this is part of the ABI [SCO97]). While most of the code will be compiled in the same way as it would without PIC (this is possible because on i386 the compiler will use EIP-relative calls for functions and store local variables or arguments on the stack without any direct addressing [Int14a, SCO97]), code that accesses global variables has to be changed in this mode.

These variables depend on their absolute address, but because of PIC the linker doesn't know where in the address space the code will be placed during runtime. This means, it also can't know the absolute addresses at link-time. Instead, the compiler will generate code that accesses these variables indirectly through a lookup in the GOT. In there, the linker stores the distance of these variables from the beginning of the code — in the same way as the addresses in the used position independent address space are calculated. Before the stub can use this table then, it has to add its start address to each of the fields in the GOT (this address is calculated as explained in 5.2.1 on page 68).

After this is done, the stub can make use of otherwise unaltered code, compiled with a normal C compiler following the *System V i386 ABI* [SCO97].

Create Jailhouse's Page Table The stub will now setup the page table necessary to switch from the current protected mode to the target IA-32e operating mode. To prevent redundant work (Jailhouse would normally also create its own page table), the stub will create the paging structures in the exact same way as Jailhouse would do it and store them them in the same locations, too. All the information necessary for this process can be found in either the normal Jailhouse header or in the Jailhouse configuration.

At this point, the BSP also has all information necessary to locate its own per-CPU area in Jailhouse's memory space, and it can switch its stack from the private area to the one Jailhouse will later use for it as well.

Wakeup of the RLPs and Re-Activation of the SMIs The BSP has now done all necessary steps to switch the operating mode. But at this point, all the RLPs are still disabled. After the measured launch, they can not be woken with the scheme normally used to initialize a multi processor system on Intel's x86 architecture (the INIT-SIPI-SIPI sequence [Int14b]). Instead, it has to be done with another subcommand from the new `GETSEC` instruction (the subcommand `WAKEUP`).

Before the BSP can use this command, it has to setup a data structure containing: the desired EIP, desired GDT and the associated segment selectors (this is called the *JOIN*-structure). All of these are used to initialize the RLPs during their wakeup.

This makes the startup more flexible than the initial jump from ACM to the BSP. The RLPs will already have a valid memory layout and usable data segments, otherwise their architectural state is the same as that of the BSP at its wakeup (which means, they also start in the 32 bit unpaged protected mode). The address of this struct has to be written to the TXT config space and afterwards the BSP can call `GETSEC[WAKEUP]` to proceed.

During this wakeup, the BSP will wait till it receives a signal from at least as many RLPs as configured for Jailhouse — similar to before the call of `GETSEC[SINIT]`. In case more processors than this wake up, they will cause a TXT reset, because this state wouldn't match the measured configuration. In the meantime, the RLPs will start and configure the remaining parts of their environment that are not already set by the join structure. This covers their own stack — in the correct position within the per-CPU area of Jailhouse — and a working IDT.

For the identification of their respective spot in the per-CPU area of Jailhouse, the same restriction applies as when they saved their state on the TXT heap (see in 6.6.2 on page 95).

The first step all processors will do together is to re-enable SMIs. This is also the last step in the stub where SMX support is necessary (there is no other way to do this, because there is no normal way to disable SMIs) — the corresponding subcommand in this case is `SMCTRL`. A discussion of the security implications of this step can later be found in 7.5.2 on page 114.

Restore of the Saved MTRR Settings In the process of preparing the measured launch, the system's MTRRs were changed to fit the requirements set by TXT. So the Linux that will later run as guest of Jailhouse can work correctly, these MTRRs have to be changed back to their original value. But MTRRs can not be virtualized with VT-x, there is no place for them in the VMCS and thus they have to be restored without it (to not infringe upon the fidelity characteristic of the VMM). But this also means that the MLE has to use unmeasured and therefore untrusted states.

To guarantee that the MLE can perform undisturbed by any intentional or also unintentional failures in the saved state, it is necessary for the stub to check that every used memory area in the MLE is mapped with a supported caching type. In case of Jailhouse, the stub has to check at least that the configured hypervisor memory and all used MMIO spaces are mapped with a fitting type (for example, the TXT config space has to be mapped with the Uncachable type).

But unfortunately, there was not enough time to analyse this fully during this work. To not disturb the guests of Jailhouse with potentially hard to debug behaviour, it was therefore decided to re-apply the saved states without further checks. To ensure the correct settings here, further work in this area is necessary.

One important guarantee that the Intel manual gives however, in case of an overlap between MTRRs and page-level control, is that whatever control restricts caching more becomes the effective one [Int14b]. So for example, in case the saved MTRRs map a range of memory with more caching than the Jailhouse page table would provide, then the settings of the page table would be used. In the opposite case, it would only lowers the performance, but not the function (all the source code written for Jailhouse and the stub is designed to work with the most caching enabled).

Pre-Initialization of Jailhouse and the Final Jump All other saved states on the TXT heap (the individual architectural states of the processors) are not re-applied during the execution of the MLE. They will only be re-applied once Jailhouse is completely configured and starts the VMX non-root operations. But by then, it also has secured itself sufficiently (for example, through the use of the IOMMU and EPT mappings).

The only remaining task for the stub with those states is to copy them from the TXT heap into the per-CPU area of Jailhouse. Each processor will copy its own state from the heap into the same position within its per-CPU area where Jailhouse would put it too, without being launch with TXT. This step can then be skipped in Jailhouse.

With this step the function of the TXT stub is done. It can now make the operating mode switch to IA32-e and then jump to the normal Jailhouse code. The final jump location is the same entry point that the normal Jailhouse module would use, only with some different function arguments to distinguish both from each other.

6.5 Changes Made to the Jailhouse Hypervisor

Because of the design and the pre-initializations that the TXT stub already does, changes to the hypervisor code itself could be kept to a minimum.

After the entry point of Jailhouse is called, it is now necessary to distinguish between whether the Linux module called it directly, or if it was target of the TXT stub. In the later case, it will use a slightly altered startup phase and store the information about it in Jailhouse's core (a flag, telling that the measured launch was made). Later, when the hypervisor is about to be shutdown again, it can look up this information and decide whether it needs to use the TXT variant or the normal shutdown.

The Altered Startup In the course of the Jailhouse startup phase, after it was called by the TXT stub, the core doesn't need to save Linux's information anymore. This was already done during the TXT setup, and later in the TXT stub those information got copied into the per-CPU area of Jailhouse.

The same is true for Jailhouse's page table. While Jailhouse would normally replace Linux's page table with a new and self-created one, it can now initialize its paging code with the page table that the TXT stub created. At this point, this table will contain mappings for the whole hypervisor memory space. Any further mappings that are necessary have to be done without any changes to the startup code.

Apart from these two changes, the Jailhouse core will execute unaltered. Its final action during the startup phase is to fill the VMCS's guest-state area with the stored information of the previously running Linux and to enable the VMX guest operations.

Ending Jailhouse After a Measured Launch Should the measured hypervisor ever be shut down again, it has the responsibility to ensure that no other software can impose as it or use secret information of it.

The decision whether a user or a remote host thinks that Jailhouse is running, is based on the measurements in PCR 17 and 18 of the system's TPM. Jailhouse has to ensure that these measurements don't remain the same after it has shut down. Other than that, there are currently no secrets kept by Jailhouse, they don't need to be protected.

To ensure the change of the measurements stored in the TPM, Jailhouse can make use of the same technique as Flicker does (see 4.2 on page 55): it extends a well-known value into at least one of the used PCRs.

This approach could not yet be used in the current implementation for this work. It would require TPM functionality that could not be implemented within the time constraints of it. Instead, the current implementation does a TXT reset whenever Jailhouse is about to shut down. The following system reset will reset the PCRs as well. This has to be improved in future work to complete the implementation.

Other than that, after the stub has extended the PCRs, a future implementation also has to shut down TXT properly. This is done with the SMX instruction `GETSEC[SEXIT]` and is far less complicated than `SINIT`.

`GETSEC[SEXIT]` has to be called on the BSP and outside of any VMX operations (the hypervisor has to finalize this first). The system will again gather all processors during this call, but it will not reset their state again. Instead, it will only reset internal flags and otherwise return the processors in the same state as they were before being gathered. On the chipset, the processor will close the private TXT configuration space and if still opened, close locality 2 and 1 (leaving only locality 0 available to the system). After that, it will return control to the software executing on the BSP.

And with this final step, Jailhouse will have *completed the trusted execution of the hypervisor*.

6.6 Open Issues and Constraints

In general, the current implementation follows the process designed in the previous chapter. Most exceptions to this were already mentioned during the description of the implementation itself. This section will relate on the two biggest remaining problems that were found but could not be completely solved yet.

6.6.1 Fixing the MLE Image

The first of these problems is caused by limitations of the x86 architecture and thus likely to stay. As explained in the implementation of the initial phase after the measured launch (see 6.4 on page 88), in order to use its own segments and to be able to have write access to the memory, it is necessary for the stub to replace the global descriptor table (till this is done, the stub can *not* write to any memory location). This is done with the instruction `LGDT`, which in turn takes a memory operand as argument [Int14a]. At the pointed memory location it expects an address and a limit.

The address will tell the processor the *physical* location of GDT and with the limit it gets to know how many descriptors are stored in it. Calculating the limit at the compile time of the stub is simple, it is in fact a constant. But calculating the physical location at the compile or link time is not possible, it is only just known at runtime. And because it is not possible to write to the memory before the new GDT is set, it is also not possible to write the actual physical location at the runtime — rendering this whole operation without further changes impossible.

To solve this issue, the setup fixes the hypervisor image after it has been loaded by it. At this point in the setup, it already knows the Jailhouse configuration and hence the physical start address of the hypervisor memory. Using that address, it can calculate the position where in the TXT stub the GDT is located and write this address into the image at a known position. To prevent this from breaking the security concept ("no trust in data supplied from unmeasured sources"), this exact same thing has to be done at the time the hash for a trusted MLE is computed (with the configuration as input, the hash-calculator can do the exact same thing as the setup and fix the image before taking the hash).

Currently, this concept is also used for a few other variables that are hard to calculate at runtime (e.g.: the number of processors the hypervisor expects, to calculate the size of the per-CPU area to find the configuration), but ultimately those are not limitation by the platform itself.

6.6.2 Different Processor IDs

The other problem in the current implementation is another issue caused by a lack of time, but because it is security relevant, it has to be noted here.

In the design, before the setup ends and enters the measured launch, each processor saves its architectural state on the TXT heap (see 6.3 on page 85). The concept is to save this state in an associative array with the processors initial APIC ID as index. This ID can always be read with `CPUID` and can not be changed by software, and thus it can be used across the measured launch without a security risk.

But this would required further changes to Jailhouse. At the time of this writing, Jailhouse reuses Linux's logical IDs to identify processors and to identify each processor's location in the per-CPU area. This means, at the time the TXT stub pre-initializes this area and enters Jailhouse, it has to use these IDs too — and at this point it differs from the concept.

The probably quickest solution for this problem is to store the mapping between initial APIC ID and logical ID on the heap too. This way, each processor can lookup its logical ID after it has been waken up in the MLE, and thus it doesn't have to break with Jailhouse's concept.

This is the solution used in the current implementation. The problem is, this information is not measured and an attacker could change the logical IDs before the launch, making this a potential attack on the MLE!

To solve this problem, the envisioned solution is to include the mappings into the Jailhouse configuration — and with that into the measurement. Initial research for this solution has indicated that Linux's mappings between processors and logical IDs are stable across resets, and thus they would support this solution.

6.7 Code Size

To be able to better quantify the effort necessary for the implementation shown in this chapter, this section will present the size of the added and changed source code as a basic metric.

One of the main motivations for the use of the dynamic chain of trust is to make the TCB of the system smaller. This in turn is one of the motivations to use TXT

	Lines of Code	
Component	Without TXT	With TXT
Hypervisor	7368	9759
,→ TXT Stub	N/A	2188
Kernel Module	851	2339
,→ TXT Loader	N/A	1405
Tools	1013	2373
	The completely separate TXT Tool:	1360

FIGURE 6.2: Table showing the lines of code necessary for the Intel TXT implementation in Jailhouse. The presented numbers were computed with the tools `sloccount` [slo] and `git` [git] and show only lines containing functional source code | no comments, no blank lines, no build scripts and no con guration.

in Jailhouse, rather than the static chain. It is therefore not desirable to enlarge the existing code base of Jailhouse by much — especially the hypervisor core.

The size of the TXT implementation in Jailhouse, in lines of code, can be seen in Table 6.2.

As shown, the hypervisor core increased its size by 2391 lines of code — of which 2188 are added by the TXT stub and the remaining 203 as changes in the core itself. When compared to the original size of 7368 lines, this is a high number — an increase of around 32:5%. This code is not yet in a finished state and might undergo subsequent refactoring, but it also still has missing parts. The size is therefore not likely to shrink.

The same holds true for the kernel module. Its size more than doubled by around 174:8%. And while the size of the module is not as critical as the size of the hypervisor core, it still represents a large increase of source code that needs to be maintained.

Both represent the complexity of the measured launch and the many requirements it imposes on the software which wants to use it — especially the multiple mode switches, the large amount of predefined structures and the necessary page table algorithms. A more detailed overview over these can be seen in Table 6.3 on the facing page.

The numbers about the various tools are the least critical that are shown here.

Component	Lines of Code
Assembly for Mode Switches	≈ 370
Definitions for TXT in Header Files	≈ 850
Paging Implementations	≈ 350

FIGURE 6.3: Table showing a more detailed listing about the largest parts of the TXT implementation in terms of source code. These are not separated between In-TCB and not In-TCB, but only a part of the paging is not required in the TCB itself ($\approx 120 LoC$).

They complement the implementation: make it easier for the user to compute the MLE's hash, create LCP data and decode error codes. But they are not vital to the function of the measured launch itself (and even with that true, much of the TXT tool's source code is the result of the large amount of error-messages stored in it for the decoding).

6.8 Performance

The added performance overhead by the measured launch is not as important as the added code size for the presented use case in this work. A hypervisor like Jailhouse will likely only start once in a system's runtime. But to be able to derive estimations for other use case of TXT and to summarize the implementation, this section will present a comparison between the startup time of Jailhouse with and without support for Intel TXT.

The presented measurements are taken over the time the systems spends to *enable* Jailhouse. This only includes time spend after the IOCTL was already sent to the kernel part of Jailhouse, and this in turn is about to run the corresponding enable function — time spend outside of the kernel, in any involved userspace application, is not measured. The measurement ends upon the finish of this enable function.

During this time, the measurements are further divided into the main parts of the procedure: Jailhouse kernel driver, TXT loader, measured launch (including the ACM), TXT stub and Jailhouse core. The parts of this list which are also part of the start procedure in case it is run without TXT were also measurement in their unmodified version.

To take the measurements, the x86 instruction `RDTSC` was used. This returns the value of the processor's time-stamp counter at point of the instruction's execution. The *TSC* is not influenced by the measured launch, and on modern processors (Pentium 4 and following [Int14b]) it is incremented at a constant rate, unrelated to the actual clock set for the processor. But it doesn't represent an immediate time value in seconds or something similar, for this, it first has to be mapped.

To create this mapping, a first series of measurements was taken on the target platform (`Intel Core i7-4770S` at $3.10 Ghz$, $8 Gb$ RAM, Gentoo Linux with kernel `3.18.0-rc5`): in one microsecond the TSC is incremented on average by a value of $3,092.94$. This average was computed over a set of 601 measurements, each taken with an interval of one second. The standard deviation over these measurements is only 0.062, resulting in a 95% confidence interval of ± 0.0024 per measurement. Because this interval doesn't even represent a nanosecond difference per measurement, and the overall time of the launch is also influenced by disk activity, the presented average is assumed to be the exact value.

With this rate established, the launch of the hypervisor — divided into the named parts — was measured forty times, twenty with and twenty times without the TXT extension. Because the shutdown of Jailhouse with TXT support is not completely implemented yet, it was necessary to reset the system after each of the measurements, resulting in a cold launch each time. The final results can be seen in Table 6.4 on the next page.

As can be seen, the extensions made for Intel TXT and the measured launch have made for a significant increase in runtime of the hypervisor launch. And while the extension-code can still be optimized (for example, the current implementation of the TXT stub executes with *no* caching enabled), it can be seen that the measured launch alone nearly doubles the time consumed by the unmodified Jailhouse kernel driver and the core together. It can thus be assumed: unrelated to any further optimizations, the time of the Jailhouse launch with Intel TXT will at least be twice as long as it takes Jailhouse to start without TXT.

Further and more detailed measurements revealed that much of the time spend in the kernel driver and TXT loader is spend on disk activities (loading the image, configuration, ACM and LCP data). The major part of the TXT stub's runtime on the other hand ($1,388.66 ms$ with a 95% confidence interval of $\pm 0.0333 ms$) is spend on the creation of Jailhouse's paging. This time may represent the biggest possible

Component	Average (ms)	Std. Dev. (ms)	95% Conf. Inter. (ms)
Jailhouse Kernel Driver	37:31	8:836	±1:877
,→ Without TXT	32:95	0:101	±0:021
TXT Loader	38:45	6:975	±1:482
Measured Launch	856:15	4:114	±0:874
TXT Stub	1; 401:48	0:157	±0:033
Jailhouse Core	427:53	2:278	±0:484
,→ Without TXT	458:65	2:281	±0:485
The Whole Process	2; 760:92	10:969	±2:330
,→ Without TXT	491:61	2:308	±0:490

FIGURE 6.4: Table showing the measurements of the launch of a Intel TXT enabled Jailhouse, compared to a unmodi ed version. The shown results were computed over a set of 20 measurements for each version, each measurement resulting in a complete system reset before the next.

optimization potential for this whole process and should be investigated further in future work.

7 Security of the Trusted Hypervisor

With the presentation of the design and implementation of the trusted hypervisor in Chapter 5 on page 61 and Chapter 6 on page 75, it was shown what steps are necessary to make use of the measured launch with Intel TXT.

After the launch, any remote party with knowledge of the system in question can request evidence, in form of a TPM quote (see 3.3.1 on page 36), to verify the state of the system and gain trust in it — deciding whether a trusted hypervisor was launched and is still running, or not. To further increase this trust, it is now necessary to show that the presented work is secure and can prevent forging of results in the quote.

This chapter will present an analysis of the provided security by the design and implementation. It will present possible attacks on the measured launch and show how the presented system can defend itself against them. Most of it will heavily depend on the security of the TPM chip and the proper integration of it with the other involved hardware.

Due to a lack of time after finishing the design and implementation, this analysis could not be done using already established methods for security and risk analysis (examples can be found in [NA12, Nis12, BRC12]). This might present a subject for further work on this topic.

First of all, in Section 7.1 on the next page, the assumptions made for this analysis will be presented. Following them, in Section 7.2 on page 103, the same will be done for the potential attacker of the system — what motivations does he have, which potential methods and what knowledge. The discussion of the possible attacks in this framework is afterwards split into three parts. The first part, in Section 7.3 on page 104, will consider attacks involving the hardware of the system. After this, Section 7.4 on page 108 will discuss attacks via the software of the system, before the launch and afterwards. The final section is 7.5 on page 113, and will show the limitations of the system that were found in this work.

7.1 Assumptions About the Analysed System

The base of this work is that the user trusts the system's hardware — at least the hardware that is involved in the measured launch. This means, the first assumption has to be that the processor, chipset (including any built-in IOMMU), main memory, the TPM and the bus system involved to connect these are behaving as specified — an illustration of the involved components can be seen in Figure 3.7 on page 45. Especially the management of the access to the TPM's localities has to behave like specified in both the TPM spec. [TCG05] and the SMX spec. [Int14a].

Furthermore, it has to be assumed that the TPM was properly initialised before any attack can happen. The owner of the system was able to take ownership of the TPM, and with that create the EK and SRK. Following that, he was able to create at least one AIK and transmit the public portion of it unchanged to all relevant remote parties (including himself). Finally, he was able to create and fill the for Intel TXT necessary NV index, so that it may only be possible for either himself or a trusted party to change the stored information in it.

As for the involved software, namely the ACM and the launched hypervisor, the same has to be assumed: they have to behave in the intended and trusted way. For the ACM this means, it will behave like specified in the TXT specification [Int14c]. It will measure the correct values of the configured MLE into the TPM, check upon the specified error conditions and put the processor into the correct state, before handing over the control *immediately* to the configured MLE. The hypervisor part of the MLE in turn will be capable to effectively protect itself from any corruption, until either the system resets (and with that the TPM) or it ends its operations orderly. For this it makes use of the shown techniques in Chapter 2 on page 5.

There are no assumptions about the guest software of the hypervisor or about the software running before the measured launch — at least once the TPM is initialized like described above. In case of a difference from this for a specific attack, it will be explicitly stated in the description of the attack.

Lastly, during the first two sections of this chapter (7.3 on page 104 and 7.4 on page 108), the questioned trust relation has to be between the MLE and a remote party (either the user, or any system independent from it, connected via a network). This relation can not originate from within the system itself — for example, the started MLE will not be able to tell if it was started correctly or not. Why this assumption is necessary will be shown in 7.5 on page 113.

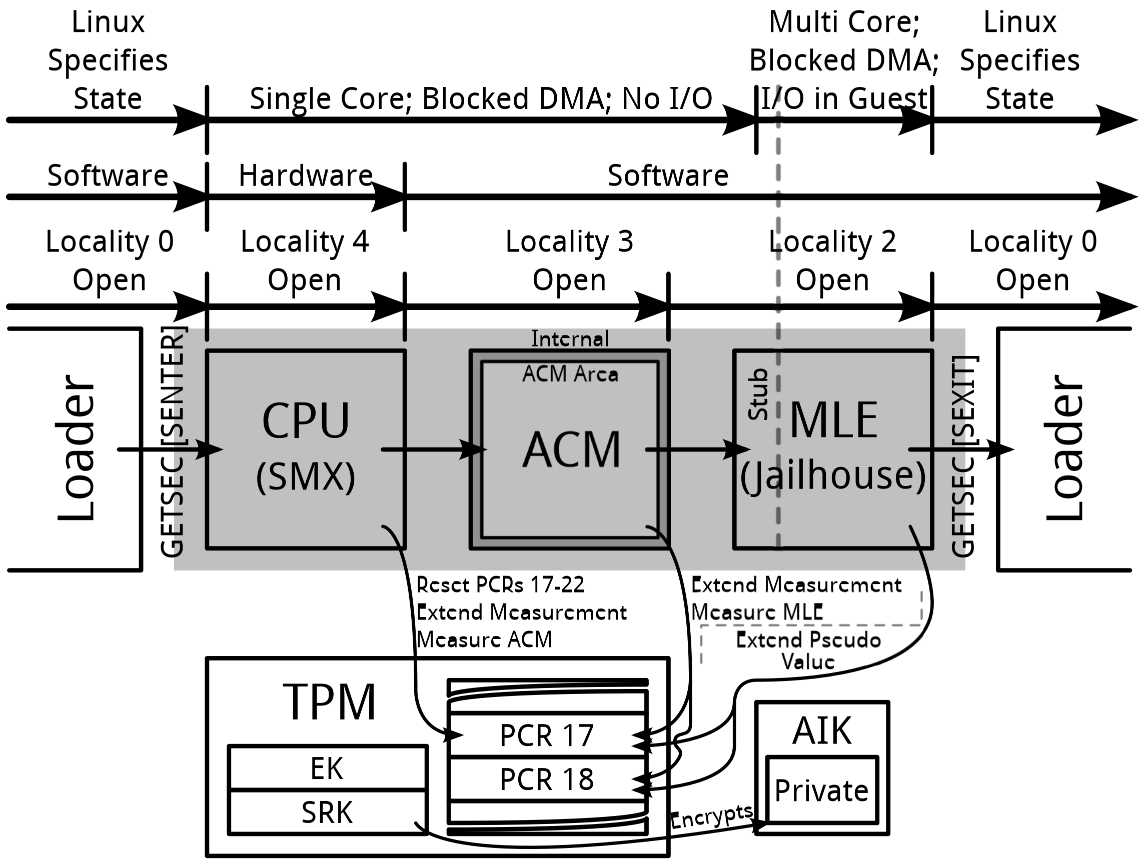

FIGURE 7.1: Overview over the analysed system, before, during and after the measured launch. The top part of gure describes, from top to bottom: active components and access possibilities to the MLE; controlling component of the measurements; the *highest*, opened locality of the TPM. The middle part shows the ow of main parts of the presented design during the main phases of it. And at the bottom, it is shown which component takes whose measurement.

Overview Over the Assumed System To get a better overview over this situation, and to wrap up all the details that were described one by one in the previous chapters, please refer to Figure 7.1.

7.2 The Attacker Profile

The prime target of an attack on the system is to influence it in a way, that upon a request for a remote attestation of its state, it will be able to send *convincing evidence* that the hypervisor in question is running on it, with the expected configuration and therefore expected guarantees, although in reality it is not. That may mean, it started the wrong or a changed hypervisor, the hypervisor is not running on the correct hardware (but again in a virtualization or emulation) or it is not running at all, but still reports so. Each of those will affect the relation between the system in question and the trusting party, and will lead to wrong assumption, which in turn may lead to unjustified discolour of secret information or surrender of control over critical systems.

To mount such an attack on the system, the attacker can use software running before the trusted launch or afterwards, from within one of the hypervisor's guests. He may also have access to the operating system's kernel and even to the module doing the setup for Intel TXT.

Physically he has access to the system's hardware. He can insert or exchange add-in cards and components of the setup (if he exchanges the processor, he may only exchange it for another trusted one). But he doesn't have the tools or knowledge to exchange any parts that are permanently mounted on the motherboard — like for example the TPM chip. Otherwise, he can only mount simple attacks on the hardware itself (as the TPM specification [TCG05, TCG11a] requires, too) and has either not the equipment, or the knowledge to mount sophisticated attacks on the bus system or any of the involved components.

Lastly, he doesn't possess any of the involved secret keys (in case of an asymmetric key, the private part) prior to a successful attack on the system, and he doesn't possess any knowledge or the required time to break one of the involved cryptographic algorithms (SHA-1, RSA with 2048 bit keys, the Diffie-Hellman key exchange).

7.3 Hardware-Based Attacks

Despite being reduced to simple attacks, a potential attacker is still able to mount the attacks discussed in this section.

7.3.1 Malicious DMA Devices

Because the attacker has physical access to the hardware, it is a simple operation for him to install additional add-in cards or exchange existing ones. If those new cards or other devices have DMA bus mastering capabilities, they are able to initiate access to the main memory at any given time during the runtime of the system, and write or read information from there without interception by the operation system. They may use this ability to change the hypervisor code after it was already measured.

The countermeasure against this type of attack is the same that was introduced in 2.2.2 on page 12 and 3.4.2 on page 42: a properly programmed IOMMU and the properties of the `GETSEC[SENTER]` instruction. In the presented architecture those are applied at two points during the measured launch.

At first, the ACM (whose memory is additionally protected by the DPR) is loaded by the processor into an internal area, designed to exclude any external and internal agents other than the BSP running `GETSEC` [Int14a]. This is done before it is measured and is kept up during its execution.

During that execution, it will check upon the second protection: the PMRs. Before it measures the MLE, it will check that it is covered by at least one of those ranges and thus protected from any DMA during and after the measurement (till the MLE itself decides to drop this protection). Furthermore, it will also check that its external counterpart and the TXT heap are properly protected by the system's DPR. Should any of these checks fail, it will mean an instant system reset (a TXT reset).

Together they guarantee that during and after the measurement no malicious device can influence the measured data — the later reported PCRs 17 and 18 will contain measurements of DMA protected memory areas. Because the ACM also checks that the DPR protects the heap, no DMA agent can influence the data the ACM puts there — the other parts of the heap, like the "SINIT to MLE" area, are still not trustworthy.

Any change before this protection was in place may be done at will, but will be measured as well by a trusted ACM. This may therefore only result in a denial-of-service by forcing the system into an untrusted state.

7.3.2 Change of Firmware Settings

Attackers with physical access to the platform can reset the system's firmware or change settings of it. They may even use the BIOS to reset the TPM (this is allowed by the specification for maintenance).

If the state of the system's firmware is of interest for the trusting party, then they have to request the states of the PCRs 0–3, as well as those of the PCRs 17 and 18. Those are specified to contain the relevant settings of the BIOS and its configuration, and the same for any option ROM. It is also not possible to reset those PCRs without a system reset [TCG12].

The reset of the TPM is more problematic, but will also at most result in a denial-of-service of the system in question. Upon a TPM reset, the SRK is permanently deleted from it. This SRK in turn was used to encrypt the private part of the AIK that is used to sign the measurements. If the SRK doesn't exist anymore, this private part can not be decrypted anymore and is rendered useless, implying that it also can't sign any measurements anymore and thus can't convince any remote party of the authenticity of a sent measurement.

7.3.3 Attacks via Power Management

Systems that possess the ability to enter different power states may be abused by entering a sleeping state where both the hypervisor's and the hardware's protection aren't active anymore, and thus, can not protect the measured environment from being changed.

On x86, the most problematic state changes are ACPI's S-state transitions [Int14c]:

- **S1**: in this state the protections will remain in place.
- **S2**: is not supported by Intel.
- **S3–S5**: these are problematic and may allow changes via hardware access after the system has been put to sleep.

To prevent the system from being put to sleep into a vulnerable state, it can either use a feature provided by TXT capable chipsets or otherwise by the VMM. Next to the registers described in Table 6.1 on page 77, the *private* TXT config space also contains a *Secrets* register. In case the MLE writes to this register, the chipset will remember this write and during an attempt to enter S3–S5, it will instead reset the system — resulting in either a denial-of-service or a reset attack (see next subsection). The alternative to this would be to use the hypervisor, it can setup its guest page tables 2.2.1 on page 8 and the IOMMU 2.2.2 on page 12 in a way to disallow any access to the relevant ACPI tables.

Those tow methods will prevent software from transferring the hardware into the problematic power states and thus prevent the MLE state from being changed.

7.3.4 Reset Attacks

The last attack discussed in this part is the possibility to reset the system. This very simple attack can easily be done by anyone that has physically access to the system — or with software means, as described in the previous attack — and it won't influence the measurement of the hypervisor directly. Instead, it can be used to attack badly designed protocols.

When a remote user starts the process of remote attestation with the system in question, it will get a signed measurement in return (see 3.3.1 on page 36). With this, he can verify the state of the system at the moment the measurement was signed. By additionally sending a nonce in the original request, this signed measurement is also guaranteed to be fresh. But only fresh at the time of the signature. After the requester has received the reply, the attacker can reset the system and boot another software of choice. The requester may still think at this point that the software, whose state he has verified, is still in control.

This kind of attack can not be prevented solely by using Intel TXT. Instead, the communication protocol has to be improved to counter it. Possibilities for this have been researched already [STRE06, GPS06, MPP+08].

The basic idea of those solutions is to create a secure channel in combinations with the reply to the request for remote attestation. McCune et al. propose the use of asymmetric encryption for this purpose:

1. After the MLE has been launch, it will create a fresh RSA key pair — it may use the TPM for this. It will then extend the public part of this key into an unused PCR, which is only accessible by it itself (in the present architecture it might use PCR 19–22 for this, those are already protected by their locality modifier). Otherwise, it will keep at least the private part secure (might be accomplished by the same mechanism securing the VMM).
2. At the time the remote user requests attestation (together with a fresh nonce), it will include this PCR into the list of PCRs signed by the quote. The TPM will thus not only sign the PCR 17 and 18 — warranting for the state of the system — but also the hash of the public key for this sessions.
3. In the reply to the request, it will then send: the PCRs 17, 18 and the PCR containing the key hash; the quote signed by the TPM (the signature will contain the same set of PCRs and the nonce); and the public key whose hash is signed by the quote.

4. The requester can now verify that the PCRs contain a valid measurement of the ACM and MLE, and it can verify that the received public key belongs to the system that sent the quote.

5. All future communication between those two will now be encrypted by those two keys.

This protocol is not perfect yet, it is secure in the communication direction from requester *to* the trusted system, but it shows the principle: if the signed quote contains a valid measurement for PCR 17 and 18, then the software on that system is trusted. In turn, it can be trusted to handle and secure the PCR that contains the key-hash correctly, together with the key itself. It is required to never safe the private key, unless properly secured, so that upon a reset or shut-down of the software or system, it will be lost and a new session has to be established. This new session will not be able to decrypt any message send with the key-pair of the old system and the protocol can react to that properly, rendering the reset attack ineffective.

Alternatively to using RSA, and only exchanging a single key, the protocol might exchange key factors for the Diffie–Hellman key exchange [DH76], and with that make the communication secure in both directions [STRE06].

7.4 Software failures and attacks

Concerning the software of the target system, an attacker is far less restricted. This section will present the considered attacks in three parts: attacks before the launch, during it and attacks against the MLE from within one the hypervisor's guests.

7.4.1 Software Attacks Prior to the Measured Launch

Before the measured launch has happened, a potential attacker has complete access to the system and is unrestricted in the privilege level he might use. As he has physical access, this might be as easy as booting his own operation system from a portable media.

Attacks Against Long Time Storage Attacks against data that are stored on the system's hard drive, or other long time storage systems, can corrupt either the stored MLE image (including its configuration), the ACM or the AIK.

All of them can at best lead to a denial-of-service of the system. The MLE and the ACM are measured, and any change to them would also be propagated to the PCR values. Because the used cryptographic hashes are collision and pre-image resistant, it is infeasible for the attacker to change the MLE or ACM in a (useful) way and still keep them resulting in the same hash value.

As for the AIK: its private part is encrypted by the private part of TPM's SRK. This is specified to never leave the TPM (not even encrypted). Nearly the same holds true for the AIK, its private part might never leave the TPM unencrypted (it is a non-migratable key [TCG11a]). Both ensure that an attacker might never decrypt the AIK and gain knowledge of what the private part is, even if he has access to its stored version. Hence, he can not forge a signature made with it and might only destroy it, so it can not be used later on.

Attacks Against the TPM Attacks against the TPM without the owner's secret are covered by its design. Most important for the presented system is that the NV index, used to store the LCP, is properly write-protected (only for the owner or a trusted other party). Otherwise an attacker might change it.

But even then, using a changed LCP might only result in a denial-of-service. It does not influence directly what the system stores into the PCRs, and thus any change will also propagate into the PCRs and will later be visible during the attestation.

Other than that, it is important to re-iterate that a software attack must never have control over locality 4 (as is specified in the TPM specification [TCG11a, TCG05]). If that is not given, then an attack might reset the dynamic PCRs at will and extend values into them, although the software represented by those values has never run. After that, the remote attestation would confirm those values and the trusting party would assume that the software is running, although it is not.

Attacks Against the Setup The last possibility consider here are attacks against the setup of the measured launch (described in 6.3 on page 80).

Because the attacker's first target is to corrupt the MLE in way, so it will behave in an untrusted manner, although the remote attestation gives evidence against that, he might not corrupt the setup in a way that makes it fail (which again will only result in a denial-of-service, because the launch will reset the system). This implies, he has to use a trusted ACM, a trusted MLE (including the Jailhouse configuration) and he has to do the setup in way that makes it launch. He may modify the processor's state, values on the TXT heap (unless this will prevent the launch from happening), the MLE page table and he may program other devices in a malicious way.

Changes of the processor's state are undone during the measured launch, regardless of their intention. After the launch, the processors will always contain the states set by the ACM. Only the MTRRs are different. They are not changed, but they are specified to be checked by both the `GETSEC[SINIT]` call and the ACM afterwards — making an invalid change impossible.

The TXT heap on the other hand may be changed undetected. From its four sections — "BIOS Data", "OS to SINIT", "OS to MLE" and "SINIT to MLE" — only the first three may be changed, the last section is created by the (trusted) ACM and may thus not be influenced without a malfunction in the ACM.

The first of them, "BIOS Data", is required to be initialised by the BIOS, but may be changed afterwards. In the presented MLE though, it is not used and changes will not have influence (interesting values in this section are covered by the Jailhouse configuration).

After that section, the heap contains the "OS to SINIT" section, which is used to configure the ACM, foremost to state what MLE shall be measured. Consistent changes here would cause the ACM to measure a different MLE, and hence would not result in the trusted PCR values. Any inconsistent change is specified to be noticed by the ACM [Int14c] — this includes the configuration of wrong PMR settings and the configuration of the wrong LCP data (whose signature is stored securely in the TPM). Thus, a change here would either result in a system reset, in case they are wrong, or in the launch of a different MLE, both failing the target.

The last modifiable section, "OS to MLE", is also the most interesting. It stores the data given by the setup to the MLE,and is not checked by any other component. In the presented design, it contains the stored MTRR settings and the architectural state of the processors before the launch.

For the architectural state, the MLE only checks if the data structure is correct (an associative array, with one entry per processor id; see 6.6.2 on page 95 for open issues concerning that topic). The MLE does not however apply any of these states, but will only use them once the hypervisor launches the VMX guest operations, and then the MLE will not be affected directly, but only the guest. This might in turn, again, lead to a denial-of-service, but will not change the MLE undetected.

As stated before, it was not yet possible to fully analyse what the MTRRs might affect during the MLE execution. Please refer to the discussion in 6.4 on page 91 for more details on this topic. Future work here is mandatory, but it was also not possible to find a useful way of how this might be successfully abused.

Like stated before, the only remaining heap section, "SINIT to MLE", is created by the ACM, and thus does not allow any influence. Attacks through malicious programming of DMA devices are handled in the same way as the attacks discussed in 7.3.1 on page 104. Other external events that can be programmed are also blocked by the changes applied during the operation of `GETSEC[SENTER]` — at least till the MLE can protect itself properly against them. Leaving the attacker with only the MLE page table as surface.

In this case, he at least has to follow the rules set by the ACM, otherwise the system would reset. By doing so, he could map a different MLE, but this would be noticed during the remote attestation, because the measurements would be different. This means, he has to map the correct MLE, in order for the correct measurements to be taken. The only remaining possibility is to change the mappings between linear and physical address into an unexpected physical layout. This was already discussed in 6.4 on page 87, and it was also shown how the MLE can defend itself against it.

This concludes the attacks that were considered to be possible before the launch.

7.4.2 Attacks Against the Measured Launch

The next logical step, after the setup is done, is the measured launch. Software attacks here are generally prevented by the design of `GTESEC[SINIT]`. This was already described in details in 3.4.2 on page 42 and is based on the secure environment created by it.

After the setup instructs `GTESEC[SINIT]`, the processor will take over the control of the system, it will disable all processors but one, mask all external events and protect critical areas from DMA. At this point, no user defined software can execute, and after it is done, it will only hand off control to a *verified* and measured ACM. Which in turn is run inside its own protected processor area, still with all other processors disabled, and only allows code inside the measured MLE to take over control.

This make software attacks during this step impossible — they simply have no processor to execute.

7.4.3 Attacks Against the Measured Launch Environment

The last possible attack surface is after the MLE has launched. This is not covered by guarantees made by Intel TXT, but has to be done within the MLE. In case of this work, it was assumed that the started hypervisor is able to protect itself and its guests properly (by using, for example, SLAT or IOMMU protection) — which in turn is not part of this work, but is also actively researched.

By not using any external information, but only the ones given to it by either the ACM or by the (measured) hypervisor configuration, the TXT stub lays the ground work for this. This way, it can not be influenced by any other software (DMA is still blocked by the PMRs and external interrupts or NMIs are not enabled before the beginning of the VMX guest operations).

For the hypervisor itself, there are three new areas to protect, after it has been launched with TXT. It needs to protect the private TXT configuration space, the opened locality, and it needs to prevent other software from imposing as the hypervisor.

The TXT configuration space is programmed via MMIO (see 6.1 on page 76), and this is not filtered any further by the chipset. Any software that has access to the physical address range of the private space can also program TXT with it. To prevent guests from doing so, the hypervisor has to make sure that this physical address space is never included in one of its guests address spaces (exceptions may be made, but only if this is explicitly the desire of the system's operator and even then it is probably enough to only expose the public part of the space). In Jailhouse, this is done by either not mapping this memory area in any guest configuration, or by mapping it read only. Jailhouse in turn will then apply these configurations via the guests SLAT (see 2.2.1 on page 10) and IOMMU remapping structures (see 2.2.2 on page 12).

The same has to be applied to the TPM programming interface. This is organised much in the same way as the one of TXT. Each locality (0–4) has its own set of memory mapped registers that are programmed via MMIO — of which each mapping is exactly one page long and has a predefined physical start value (`0xFED40000`–`0xFED44000`). After the launch, locality four and three are closed again, and locality two is open to be used by any software that has access to its address range (starting at `0xFED42000`). To prevent unauthorized software from accessing this more privileged interface (in contrast to the always open locality null), it has to be secured with the same methods as the TXT configuration space.

Attacks whose target is to impose as the hypervisor, after this has already stopped executing, are prevented in the presented design by either resetting the whole system before handing off control, or extending a further value into PCR 17 and 18.

7.5 Known Limitations

After discussing how the presented system can defend itself against (the considered) attacks from software and hardware, this section will discuss what known limitations it has. This will be done independent from the given assumptions about the system and attacker.

Any limitations that are the result of the current implementation will not be discussed here, but were discussed in 6.6 on page 93, they are not considered to be a design flaw.

7.5.1 No Trust from Within the Launched System

The most important limitation of the presented architecture is that the software launched with the dynamic chain of trust can *not* trust itself. For example, in case of the launched hypervisor Jailhouse, it will not be able to decide on its own whether it was launched correctly with Intel TXT, or not. This decision can only be made by a remote party, such as a user or a remote system.

Let's assume a systems such as displayed in Figure 7.2 on the following page. In this scenario the hypervisor will not be launched on real hardware, instead it is launched on a (hypothetical) perfect emulation of the system. It will not be able to see any differences between the presented environment and a real system, hence it can setup and launch the dynamic chain, just as it would otherwise. Any other checks it can do after it has launched, will also behave like they would on the real hardware (for example queries to the TXT config space or the TPM).

On this system, the only thing that can not be emulated is the knowledge of the real TPM's SRK and hence the decryption of the used AIK — it might present

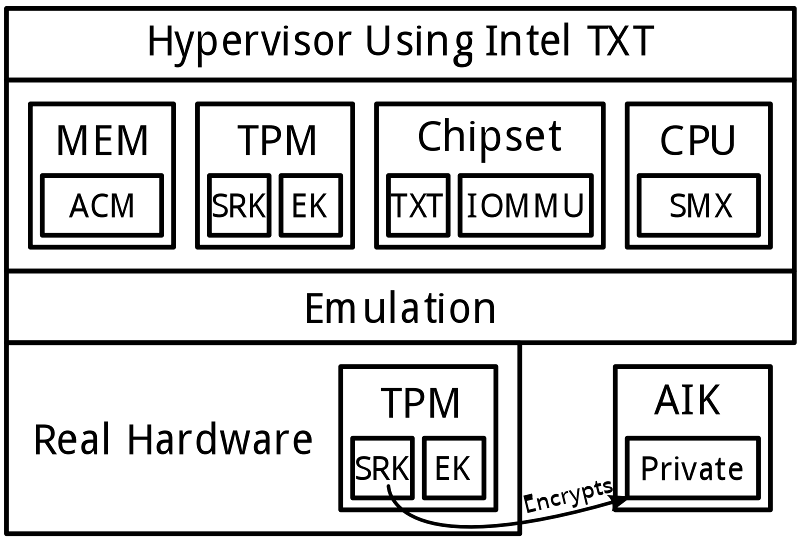

FIGURE 7.2: Diagram showing a system where the measured launch is not done on real hardware, but on top of a perfect emulation. In a scenario like this, the launched hypervisor will be unable to determine whether he is running on real hardware or not, and either will he be able to determine whether he represents a trusted con guration or not.

the hypervisor with an arbitrary other AIK, in case it wants to use it. But the hypervisor itself can not use this to its advantage.

To test whether the real TPM signs a quote with the correct AIK, or the emulation with an arbitrary other key, it has to *know* the corresponding public key of the AIK. But how can it make sure that the emulation doesn't just change the key before it is used by the hypervisor? It has to use the knowledge it is about to verify, in order to verify, whether it can trust this knowledge, or not. This is tautology, and hence it will always be exactly what the emulation wants it to be.

In contrast, even this system can not forge a quote to convince a remote verifier to trust the system. It is still not able to decrypt the AIK and hence can not make a valid signature with it. This has still to be done with the real TPM and this will only then contain valid measurements, if the measured launch was done on the real hardware.

7.5.2 System Management Mode

During the discussion of the measured launch, it was also described that the processor will mask the SMI-pin of the processor. Till the TXT stub unmasks this pin again via `GETSEC`, it is not possible for the system to enter the *System Management Mode* (abbr. *SMM*).

What exactly is done, in case this mode is activated, can not be influenced during the normal system operations and is also not known in most cases. It is part of the firmware of the system and implements tasks like power management, monitoring or any other task the firmware might see fit to execute transparently to the operating system [Int14b]. There is no obvious way to tell whether these tasks are vital to the system or not. Because of that, the MLE has to enable them again eventually.

After that, it again looses complete control over the system in case a SMI is triggered.

The design presented in this work fails to address this issue. One intuitive solution would be to argue, that because it is part of the system's firmware, it has to be measured by the static root of trust and thus can also be included in the quoted PCRs. But while this is true, the SMM has been shown to be attackable even during the runtime, failing to isolate itself properly [WR09].

A possible remedy for this problem might be Intel's *SMM-transfer monitor* (abbr. *STM*) [Int14b], but it was not possible to evaluate this feature during this work, and thus it is not possible to make predictions about its security.

This new feature is described to add a new kind of VM exist to VMX. Those would be triggered in case the VMM or its guests receive a SMI and normally would immediately enter the SMM routines. But instead of that, they would enter a separately described guest (with its own VMCS) that in turn runs the normal SMM routines. Because this is a normal VMX guest, it can be forced to follow the same restrictions and the MLE could again protect itself effectively by not mapping its physical memory in the SLAT.

But to confirm this, future work has to explore those possibilities in detail.

8 Future Work

Throughout this work, it was already pointed out that several areas will need further effort, so they can be better used and/or be better understood. Because TXT touches multiple areas of the system architecture and requires cooperation of many system components in order to work, it was not possible to explore them all in their full depth.

Concerning the task of this work specifically, there is still work open in order to complete the implementation and to fully comply with the presented design:

- At the moment, the MTRR settings, stored during the setup phase of the measured launch, are restored unchecked. In order to not compromise the MLE, it is necessary to implement checks which at least cover all the measured areas and all the resources that are later allocated by the hypervisor, and which test whether the areas have correct cache settings, or not. Because the stub can not know what the hypervisor will use and what not, this needs to be done in both parts of the MLE.
- The pass through of the logical Linux processor IDs needs to be fixed — MLE and hypervisor need to be independent from any data created by Linux and not measured during the launch. The envisioned solution for this was also already presented, and is to include a fixed translation table from initial APIC ID to logical ID into the hypervisor configuration, but this also needs to be evaluated further.
- Lastly, the shutdown of Jailhouse, when launched with TXT, is not yet possible (it will always reset the system instead). In order to prevent software running after Jailhouse from imposing as it, there needs to be a rudimentary TPM implementation which is able to extend values into the dynamic PCRs that represent Jailhouse.

Apart from these, the current implementation is functionally complete. Although it needs some further optimization and cleanup, it is able to stably launch Jailhouse.

More work needs to be done in order to use this solution in productively used systems. For one, the initial setup of the TPM — taking ownership, creating and retrieving the AIK and installing the TPM part of the LCP — has to be researched more. Currently, this all needs to be done by hand and from within an already installed operating system. But in case of a more widespread deployment of the presented techniques, this will have to be done in a automatable way.

The same holds true for updates of the hypervisor. But because Intel's LCP data are already designed to make this possible without any modification of the system's TPM, this is not as complex as the initial setup. It will nevertheless still be necessary to create a suitable scheme to distribute the updates of the hypervisor, its configuration and the new LCP data (signed with the RSA key whose public part is stored in the system's TPM).

Concerning the security of the presented system and the hypervisor in general, the biggest open point for further work is the SMM, and how to handle this mode in a way, so it can not influence the measured hypervisor in a malicious way — might it only be constant interrupts that slow it down considerably. Intel's SMI monitor concept might be the answer to this problem, but this needs to be researched in greater detail. Furthermore, an alternative for this on AMD platforms is not yet known — making this a very limited solution.

A similar concern for portability can also be seen in Intel TXT. While Jailhouse recently gained support for AMD processors in general, the trusted execution is only possible on Intel. Because of the limited time for this work, it was not possible to research AMD's solution in more detail as well. On a first glance, it seems to explore the same principle architecture, but with less effort necessary for the setup. Instead, the work that is done by the ACM in Intel's solution has to be done by the MLE on AMD — and hence the responsibility is shifted away from the manufacturer, more towards the using software.

A last point to consider would be to provide the guests of Jailhouse with a simple way to make use of this system, too. Currently, only the hypervisor is measured, but it might also be possible to let the hypervisor extend measurements of guests and their config into a different set of PCRs upon their load and activation. This would also required a simple TPM implementation and a maintainable way to evaluate these measurements afterwards — especially when multiple guests are involved, exceeding the number of dynamic PCRs available.

9 Conclusion

Starting this work, it was shown how it is possible on modern x86 systems to implement a Virtual Machine Manager, fulfilling all requirements raised by Popek and Goldberg. Thanks to architecture extensions like VT-x or IOMMUs, this is now possible with very little need for software intervention along the working of such a VMM.

This reduces the necessary code base and can raises the runtime performance. In hypervisor solutions like Jailhouse, both advantages are used to provide safe segregation to the guest software running on top of the VMM. Further down the line, this even makes it possible to eventually verify them according to existing safety standards, and use them to implement systems with mixed criticality on commodity hardware.

But this also means, those guarantees can only be made in case the hypervisor is intact — starting at the point where it is loaded and initiated. It becomes the single point of failure for a critical system.

It was thus explored how the concepts of trusted computing can be used to solve this important task, to verify that the correct hypervisor was loaded and started — so that one can trust it.

The TPM provides proven and well defined processes to make such a task feasible. But the legacy method for implementing this process — the static chain — does not fit the requirements set by hypervisors like Jailhouse. In this work, it was therefore decided to research the dynamic chain in form of Intel TXT. It reduces the Trusted Computing Base down to only the own software and the used hardware. Furthermore, by creating a secure and encapsulated environment on demand, bare of external influences, it makes it possible to start the chain at every point during the runtime of a system.

But it was found that this flexibility comes with the price of a complex set of requirements, set for any software that wants to make use of it. With the example of the hypervisor Jailhouse, it was shown that such a target software needs to incorporate code which can execute in two operating modes of the x86 architecture. It needs to be able to bootstrap the system offhand, with only very limited external information — information evaluated or measured during the dynamic chain of trust. Using these, it has to bring the system back into a state where the target software can execute — in case of Jailhouse this requires yet an other operating mode switch.

Despite of these intrusive prerequisites, this work was able to present a working design and an example implementation of the same for Jailhouse. It is now possible for Jailhouse to launch using TXT, extend its hashes into the TPM and test them for a match before it starts — the main task was thus solved. Using existing software stacks for TPM, this makes it also possible, via the well defined process of remote attestation, to provide convincing evidence for remote users that Jailhouse is really running on the target system.

Moreover, it was presented just how much more work these prerequisites cause for the reached solution: the code base of the hypervisor core is increased by one third and its startup time slowed down by a factor of five — the large majority spend on the forced operating mode changes.

At last, it was shown what deliberate attacks can be resisted with the presented hypervisor. Short of sophisticated attacks on the hardware and attacks through firmware modes like the SMM, it was not possible to find other working attacks during this work.

Because only deliberate attacks could be found to work against such a system, this makes TXT a valuable addition for safety critical applications. With it, they can be designed to only run in case a correct hypervisor was started, and through a second channel they can provide secure evidence for it. Random software errors may only result in a denial-of-service, which can be detected and fixed by redundant systems.

If the application scenario of the used hypervisor makes is necessary to answer the question, whether it was started with the correct basis or not, then implementing Intel TXT is a valid option.

A Implementing VMMs with Trap and Emulate

The commonly used approach to fulfill all three characteristics of a VMM defined by Popek and Goldberg (see 2.1 on page 5) is the use of a *trap-and-emulate* scheme [AA06]. Instructions of the VM are executed uninterrupted as long as they don't access sensitive states — and thus possibly violate the control characteristic by accessing resources not allocated to them. Upon executing such a sensitive instruction the processor would cause a trap — on x86: cause an exception — and return control from the VM to the VMM. The VMM can then determine the cause for the trap and emulate the effect, or set the VM into an error state if the instruction was illegal. A simple example on x86 for this is the `outb` instruction. Executed with an I/O-port as argument that is not allocated to the respective VM, it would violate the control characteristic and thus has to be trapped.

De-Privileging To achieve such traps, the VMM has to make sure that all instructions that could possibly violate the control characteristic are run with a privilege level lower than the system-level — on x86 this usually means to run them with privilege level 3 [NSL^{+}06]. This way, most of the instructions of an application, such as arithmetics, position relative control instruction and simple memory accesses, will execute as they would outside of the virtual machine. All other instructions that access important control structures, such as page table registers, interrupt handling and segmentation, will cause an exception because they are executed with an insufficient privilege level.

On x86 though, this is not possible. Instructions such as `pushf` and `popf` will access sensitive information and will fail without causing a trap, when executed in an insufficient privilege level — for example, executing `popf` on any other privilege level than null will not change the interrupt flag `IF` but also not cause an exception [Int14a]. This behaviour violates the first characteristic of fidelity.

Shadow Structures The second step to implement the original *trap-and-emulate* scheme, is to manage shadow structures for all the states that the VM can't change directly. To restrain the VMs from violating the control characteristic, structs like the interrupt- or descriptor-table have to be protected by the VMM. But in case the VM access these information, it has to be presented with the expected values, otherwise the VMM would violate the fidelity characteristic. For this it uses the shadow structures; it traps accesses to the original structures, stores the data in the VM's shadow struct and puts an appropriate values into the real structure. In case the VM wants to access the information in the structure, the VMM can present the values previously stored and thus emulate the desired effect.

This is also not possible on x86. For example, "`mov EAX, CS`" is a valid instruction under every privilege level and will load the register `EAX` with the current code segment selector. This selector will contain the current privilege level in its lower 2 bits, and this value will be higher than null in a VM, even if the instruction was executed by the VM's operation system. This is clearly not the desired effect and thus violates the fidelity characteristic.

Memory Tracing Lastly, it is not possible to trap every access to sensitive structures with only the scheme presented under *De-Privileging*. Page tables, for example, are stored in the VM's memory and are accessed with normal memory access instructions, but they are also sensitive, because the VM has to be restrained from using physical memory not allocated to it. To implement this restriction, the VMM has to shadow the page table that is used during the execution of the VM. It can do this by trapping all accesses to the register storing the top page table structure (`CR3` on x86) and by putting its own structure there. In there, it can mark all pages containing sensitive structures with a higher privilege level than the VM's and thus trap all accesses to them. The addresses of those structures can again be learned by trapping instructions (for example, trapping accesses to the `CR3` register for storing the top level page structure).

Binary Transformation To implement those three techniques, it is necessary that all instructions which access sensitive information can be trapped. This was also observed by Popek and Goldberg when they formalized virtual machines. Subsequently, they formulated and proved the following theorem:

Theorem A.1. For any conventional third generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.

By observing the examples given in both *De-Privileging* and *Shadow Structures*, it can be seen that x86 does not fulfill this requirement and thus, it is not possible to construct a VMM as defined in 2.2 on page 6 with only the classic trap-and-emulate approach.

So it may still be possible to implement VMMs on x86, developers of virtualization solutions (like VMware or Xen [BDF$^+$03]) had to invent other schemes. Most commonly used was *Binary Transformation* [AA06, Bel05]. It make use of the property already described under *De-Privileging*: most instructions executed don't access sensitive information — normal user applications that run in a VM don't access any, because they are not meant to, not even outside a VM.

Thus, if the VMM transforms all instructions that accesses sensitive information, into trapping instructions, it can again make use of *trap-and-emulate*. During the VM execution, blocks (*translation units*) of its code are fetched and all the violating instructions are replaced with explicit traps. After that, the block is executed without interference from the VMM and will return to the VMM after it has finished. This process is repeated during the whole lifetime of the VM.

B List of Figures

C Nomenclature

ABI	12
Application Binary Interface	
ACM	41
Authenticated Code Module	
ACPI	20
Advanced Configuration and Power Interface	
AIK	36
Attestation Identity Key	
API	12
Application Programming Interface	
APIC	83
Advanced Programmable Interrupt Controller	
BDF	12
Bus:Device.Function	
BIOS	18
Basic Input/Output System	
BSP	43
Bootstrap Processor	
DMA	1
Direct Memory Access	
DPA	12
Device-Physical Address	
DPR	74
DMA Protected Range	
DRTR	40
Dynamic Root of Trust of Measurement	

EK Endorsement Key	35
ELF Executable and Linkable Format	51
EPT Extended Page Tables	10
FIPS Federal Information Processing Standard	33
GCC GNU Compiler Collection	65
GDT Global Descriptor Table	86
HPA Host-Physical Address	12
I/O Input/Output	1
IBM International Business Machines	15
ID Identifier	20
IDT Interrupt Descriptor Table	86
ILP Initiating Logical Processor	43
IOAPIC I/O Advanced Programmable Interrupt Controller	21
IOCTL Input/Output Control	19
IOMMU Input/Output Memory Management Unit	1
IPL Initial Program Load	34
ISA Instruction Set Architecture	42

LCP	41
Launch Control Policy	
LPAR	15
Logical Partition	
LPC	32
Low Pin Count	
MADT	45
Multiple APIC Description Table	
MLE	41
Measured Launched Environment	
MMIO	10
Memory-mapped I/O	
MMU	11
Memory Management Unit	
MSR	8
Model-Specific Register	
MTRR	83
Memory Type Range Registers	
NV	39
Non-Volatile	
OS	2
Operation System	
PAE	46
Physical Address Extension	
PAL	53
Piece of Application Logic	
PAT	83
Page Attribute Table	
PCI	13
Peripheral Component Interconnect	
PIC	66
Position Independent Code	
PLA	68
Physical Load Address	

PMR	Protected Memory Registers	80
RLP	Responding Logical Processor	43
ROM	Read Only Memory	34
RSA	Rivest-Shamir-Adleman	34
RTM	Root of Trust for Measurement	37
RTOS	Real-Time Operating System	19
RTR	Root of Trust for Reporting	37
RTS	Root of Trust for Storage	36
SHA	Secure Hash Algorithm	34
SMI	System Management Interrupt	43
SMM	System Management Mode	112
SMX	Safer Mode Extensions	42
SRK	Storage Root Key	35
STM	SMM-transfer monitor	113
SVM	Secure Virtual Machine	1
TCB	Trusted Computing Base	29
TCG	Trusted Computing Group	32

TPM	Trusted Platform Module	2
TXT	Trusted Execution Technology	2
UEFI	Unified Extensible Firmware Interface	2
VAS	Virtual Address Space	68
VM	Virtual Machine	1
VMCS	Virtual-Machine Control Structure	8
VMM	Virtual Machine Monitor	1
VMX	Virtual Machine eXtensions	8
VT-x	Virtualization Technology for x86	1

D Bibliography

[AA06] Adams, Keith ; Agesen, Ole: A Comparison of Software and Hardware Techniques for x86 Virtualization. In: *SIGARCH Comput. Archit. News* 34 (2006), Oktober, Nr. 5, 2–13. `http://dx.doi.org/10.1145/1168919.1168860`. – DOI 10.1145/1168919.1168860. – ISSN 0163–5964 1, 7, 10, 12, 121, 123

[AJM$^+$06] Abramson, Darren ; Jackson, Jeff ; Muthrasanallur, Sridhar ; Neiger, Gil ; Regnier, Greg ; Sankaran, Rajesh ; Schoinas, Ioannis ; Uhlig, Rich ; Vembu, Balaji ; Wiegert, John: Intel Virtualization Technology for Directed I/O. In: *Intel Technology Journal* 10 (2006), S. 179–192 1, 2, 12, 13

[AMD05] AMD ; Advanced Micro Devices (Hrsg.): *Secure Virtual Machine Architecture Reference Manual.* 3.01. Advanced Micro Devices, May 2005 1, 2, 7, 40, 51

[AMD08] AMD: AMD-V Nested Paging / Advanced Micro Devices. 2008. – White Paper 10

[AMD13] AMD ; Advanced Micro Devices (Hrsg.): *AMD64 Architecture Programmer's Manual Volume 2: System Programming.* 3.23. Advanced Micro Devices, May 2013 40, 51, 80

[BCDB$^+$14] Buecker, Axel ; Chakrabarty, Boudhayan ; Dymoke-Bradshaw, Lennie ; Goldkorn, Cesar ; Hugenbruch, Brian ; Nali, Madhukar R. ; Ramalingam, Vinodkumar ; Thalouth, Botrous ; Thielmann, Jan ; Edition, Second (Hrsg.): *Security on the IBM Mainframe: Volume 1 A Holistic Approach to Reduce Risk and Improve Security.* International Business Machines, 2014 15

[BDF$^+$03] Barham, Paul ; Dragovic, Boris ; Fraser, Keir ; Hand, Steven ; Harris, Tim ; Ho, Alex ; Neugebauer, Rolf ; Pratt, Ian ; Warfield, Andrew: Xen and the Art of Virtualization. In: *SIGOPS*

Oper. Syst. Rev. 37 (2003), Oktober, Nr. 5, 164–177. http://dx.doi.org/10.1145/1165389.945462. – DOI 10.1145/1165389.945462. – ISSN 0163–5980 1, 123

[BDR+12] Bugnion, Edouard ; Devine, Scott ; Rosenblum, Mendel ; Sugerman, Jeremy ; Wang, Edward Y.: Bringing Virtualization to the x86 Architecture with the Original VMware Workstation. In: *ACM Trans. Comput. Syst.* 30 (2012), November, Nr. 4, 12:1–12:51. http://dx.doi.org/10.1145/2382553.2382554. – DOI 10.1145/2382553.2382554. – ISSN 0734–2071 1, 7

[Bel05] Bellard, Fabrice: QEMU, a Fast and Portable Dynamic Translator. In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. Berkeley, CA, USA : USENIX Association, 2005 (ATEC '05), 41–41 123

[Ben11] Bendersky, Eli: *Position Independent Code (PIC) in shared libraries*. Internet. http://web.archive.org/web/20141102134214/http://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries/. Version: November 2011 68, 69, 89

[Bin06] Binstock, Andrew: *Uses of Virtualization*. InfoWorld Virtualization Executive Forum, 2006 1

[BM13] Binu, Sumitra ; Misbahuddin, Mohammed: A Survey of Traditional and Cloud Specific Security Issues. In: *Communications in Computer and Information Science* 377 (2013), S. 110–129 1

[BRC12] Behnia, Armaghan ; Rashid, Rafhana A. ; Chaudhry, Junaid A.: A Survey of Information Security Risk Analysis Methods. In: *Smart CR* 2 (2012), Nr. 1, 79-94. http://dblp.uni-trier.de/db/journals/smartcr/smartcr2.html#BehniaRC12 101

[ByMK+06] Ben-yehuda, Muli ; Mason, Jon ; Krieger, Orran ; Xenidis, Jimi ; Doorn, Leendert V. ; Mallick, Asit ; Wahlig, Elsie: Utilizing IOMMUs for virtualization in Linux and Xen. In: *In Proceedings of the Linux Symposium*, 2006 1

[CAA08] Cucinotta, Tommaso ; Anastasi, Gaetano ; Abeni, Luca: Real-Time Virtual Machines. In: *Proceedings of the 29th IEEE Real-Time System Symposium (RTSS 2008)*, 2008 15

[CGFC10] Cucinotta, Tommaso ; Giani, Dhaval ; Faggioli, Dario ; Checconi, Fabio: Providing Performance Guarantees to Virtual Machines Using Real-Time Scheduling. In: *Euro-Par 2010 Parallel Processing Workshops*, 2010, S. 657–664 15

[Cor10] Corp, Wave S.: *Trusted Computing: An already deployed, cost effective, ISO standard, highly secure solution for improving Cybersecurity*. `http://www.nist.gov/itl/upload/Wave-Systems_Cybersecurity-NOI-Comments_9-13-10.pdf`. Version: July 2010 2, 32

[CYC$^+$08] Challener, David ; Yoder, Kent ; Catherman, Ryan ; Safford, David ; Doorn, Leendert V. ; Housley, Michelle (Hrsg.): *A Practical Guide to Trusted Computing*. IBM Press, 2008 2, 32

[DH76] Diffie, Whitfield ; Hellman, Martin E.: New directions in cryptography. In: *Information Theory, IEEE Transactions on* 22 (1976), Nr. 6, S. 644–654 108

[DWQM14] Doughty-Whit, Pearl ; Quick, Miriam ; McCandless, David: *Codebases*. Internet. `https://docs.google.com/spreadsheet/ccc?key=0Aqe2P9sYhZ2ndElxbjVTcnV2bHFOWmUwSkt2bjZLdVE&usp=drive_web#gid=5`. Version: November 2014 38

[FG13] Futral, William ; Greene, James ; Pepper, Jeffrey (Hrsg.) ; Hauke, Patrick (Hrsg.): *Intel Trusted Execution Technology for Server Platforms*. Apress Media, 2013 2, 40, 41, 53

[FIP01] FIPS ; National Institute for Standards and Technology (Hrsg.): *Security Requirements for Cryptographic Modules*. Gaithersburg, MD 20899-8900, USA: National Institute for Standards and Technology, Mai 2001. – viii + 61 S. `http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf;http://csrc.nist.gov/publications/fips/fips140-2/fips1402annexa.pdf;http://csrc.nist.gov/publications/fips/fips140-2/fips1402annexb.pdf;http://csrc.nist.gov/publications/fips/fips140-2/fips1402annexc.pdf;http://csrc.nist.gov/publications/fips/fips140-2/fips1402annexd.pdf`. – Annex A: Approved Security Functions (19 May 2005); Annex B: Approved Protection Profiles (04 November 2004); Annex C: Approved Random Number Generators

(31 January 2005); Annex D: Approved Key Establishment Techniques (30 June 2005). Supersedes FIPS PUB 140-1, 1994 January 11. 33

[FO06] Fisher-Ogden, John: Hardware Support for Efficient Virtualization / University of California, San Diego. 2006. – Forschungsbericht 10

[GCK05] Giffin, Jonathon T. ; Christodorescu, Mihai ; Kruger, Louis: Strengthening Software Self-Checksumming via Self-Modifying Code. In: *Proceedings of the 21st Annual Computer Security Applications Conference*. Washington, DC, USA : IEEE Computer Society, 2005 (ACSAC '05). – ISBN 0–7695–2461–3, 23–32 32

[git] `http://git-scm.com/` 96, 127

[GNU13] GNU: *LD Documentation*. `https://sourceware.org/binutils/docs/ld/`. Version: Marhc 2013. – version 2.23.2 67, 79

[Gol74] Goldberg, Robert P.: Survey of Virtual Machine Research. In: *Computer* 7 (1974), S. 34–45 1

[GPS06] Goldman, Kenneth ; Perez, Ronald ; Sailer, Reiner: Linking Remote Attestation to Secure Tunnel Endpoints. In: *Proceedings of the First ACM Workshop on Scalable Trusted Computing*. New York, NY, USA : ACM, 2006 (STC '06). – ISBN 1–59593–548–7, 21–24 107

[Gre13] Greene, James: *Intel Trusted Execution Technology Whitepaper*. January 2013 2

[Int02] Intel: Intel Low Pin Coun / Intel Corporation. 2002 (1.1). – Forschungsbericht 32

[Int13] Intel ; Intel Corporation (Hrsg.): *Intel Virtualization Techology for Directed I/O*. 2.2. Intel Corporation, Sptember 2013. `http://www.intel.com/content/www/us/en/intelligent-systems/intel-technology/vt-directed-io-spec.html` 13, 14, 82

[Int14a] Intel ; Intel Corporation (Hrsg.): *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2*. Intel Corporation, June 2014 41, 42, 44, 45, 89, 94, 102, 105, 121

[Int14b] Intel ; Intel Corporation (Hrsg.): *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3*. Intel Corporation, June 2014 2, 7, 8, 11, 19, 85, 89, 90, 91, 98, 115

[Int14c] INTEL ; INTEL CORPORATION (Hrsg.): *Intel Trused Execution Technology*. Intel Corporation, May 2014 2, 40, 41, 44, 46, 48, 49, 76, 82, 83, 102, 106, 110, 126

[jaia] `https://github.com/siemens/jailhouse` 15

[Jaib] *The Jailhouse Hypervisor*. `https://github.com/siemens/jailhouse` 3

[Jon10] JONES, M. T.: Virtio: An I/O virtualization framework for Linux / IBM's developerWorks. Version: January 2010. `http://www.ibm.com/developerworks/library/l-virtio/`. 2010. – Forschungsbericht 12

[Kis14] KISZKA, Jan: *Real Safe Times in the Jailhouse Hypervisor*. `http://events.linuxfoundation.org/sites/events/files/slides/ELCE-2014-Jailhouse.pdf`. Version: October 2014 3, 15

[KKL+07] KIVITY, Avi ; KAMAY, Yaniv ; LAOR, Dor ; LUBLIN, Uri ; LIGUORI, Anthony: kvm: the Linux virtual machine monitor. In: *Ottawa Linux Symposium*, 2007, 225–230 19

[Kle09] KLEIDERMACHER, David: Methods and Applications of System Virtualization using Intel Virtualization Technology. In: *Intel Technology Journal* 13 (2009), S. 74–83 1

[KNS13] KEDIA, Pooja ; NAGPAL, Renuka ; SINGH, Tejinder P.: A Survey on Virtualization Service Providers, Security Issues, Tools and Future Trends. In: *International Journal of Computer Applications* 69 (2013), May, Nr. 24, S. 36–42. – Full text available 1, 15

[Kuo12] KUO, Shih: Intel 64 Architecture Processor Topology Enumeration / Intel. Version: 2012. `https://software.intel.com/en-us/articles/intel-64-architecture-processor-topology-enumeration`. 2012. – Forschungsbericht 85

[Lov10] LOVE, Robert: *Linux Kernel Development (3rd Edition)*. 3. Addison-Wesley Professional, 2010 `http://amazon.com/o/ASIN/0672329468/`. – ISBN 9780672329463 19

[MLQ+10] MCCUNE, Jonathan M. ; LI, Yanlin ; QU, Ning ; ZHOU, Zongwei ; DATTA, Anupam ; GLIGOR, Virgil ; PERRIG, Adrian: TrustVisor: Efficient TCB Reduction and Attestation. In: *Proceedings of the 2010*

IEEE Symposium on Security and Privacy. Washington, DC, USA : IEEE Computer Society, 2010 (SP '10). – ISBN 978–0–7695–4035–1, 143–158 57, 58

[MPP+08] McCune, Jonathan M. ; Parno, Bryan J. ; Perrig, Adrian ; Reiter, Michael K. ; Isozaki, Hiroshi: Flicker: An Execution Infrastructure for Tcb Minimization. In: *SIGOPS Oper. Syst. Rev.* 42 (2008), April, Nr. 4, 315–328. `http://dx.doi.org/10.1145/1357010.1352625`. – DOI 10.1145/1357010.1352625. – ISSN 0163–5980 55, 107

[NA12] Nist ; Aroms, Emmanuel: *NIST Special Publication 800-39 Managing Information Security Risk.* Paramount, CA : CreateSpace, 2012. – ISBN 1470110598, 9781470110598 101

[Nis12] Nist: *NIST Special Publication 800-30 Guide for Conducting Risk Assessments.* Revision 1. National Institute of Standards and Technology, 2012 101

[NIS14] NIST: *Validated FIPS 140-1 and FIPS 140-2 Cryptographic Modules.* Internet. `http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140val-all.htm`. Version: December 2014 33

[NSL+06] Neiger, Gil ; Santoni, Amy ; Leung, Felix ; Rodgers, Dion ; Uhlig, Rich: Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. In: *Intel Technology Journal* 10 (2006), S. 167 1, 7, 10, 121

[OFBI10] Okuji, Yoshinori K. ; Ford, Bryan ; Boleyn, Erich S. ; Ishiguro, Kunihiro: The multiboot specification / Free Software Foundation. 2010 (1.6). – Forschungsbericht 54

[Oku12] Okuji, Yoshinori K.: *GNU Grub.* `http://www.gnu.org/software/grub/`. Version: 02 2012 54

[PG74] Popek, Gerald J. ; Goldberg, Robert P.: Formal Requirements for Virtualizable Third Generation Architectures. In: *Commun. ACM* 17 (1974), Juli, Nr. 7, 412–421. `http://dx.doi.org/10.1145/361011.361073`. – DOI 10.1145/361011.361073. – ISSN 0001–0782 6

[PMP11] Parno, Bryan ; McCune, Jonathan M. ; Perrig, Adrian: *Bootstrapping Trust in Modern Computers.* Springer, 2011 25, 28, 32, 35, 58, 59

[PP10] PAAR, Christof ; PELZL, Jan: *Understanding Cryptography: A Textbook for Students and Practitioners.* 1st ed. 2010. Springer, 2010 http://amazon.com/o/ASIN/3642041000/. – ISBN 9783642041006 29, 31

[Rus08] RUSSELL, Rusty: Virtio: Towards a De-facto Standard for Virtual I/O Devices. In: *SIGOPS Oper. Syst. Rev.* 42 (2008), Juli, Nr. 5, 95–103. http://dx.doi.org/10.1145/1400097.1400108. – DOI 10.1145/1400097.1400108. – ISSN 0163–5980 12

[SCO97] SCO: System V Application Binary Interface - Intel386 Architecture Processor Supplement / The Santa Cruz Operation, Inc. 1997 (Fourth Edition). – Forschungsbericht 89

[Shi07] SHIREY, R.: *Internet Security Glossary, Version 2.* RFC 4949 (Informational). http://www.ietf.org/rfc/rfc4949.txt. Version: August 2007 (Request for Comments) 27

[Sin04] SINGH, Amit: *An Introduction to Virtualization.* Internet. http://www.kernelthread.com/publications/virtualization/. Version: January 2004 5

[Sin14a] SINITSYN, Valentine: *Understanding the Jailhouse hypervisor, part 1.* LWN.net. http://lwn.net/Articles/578295/. Version: January 2014 2, 15, 22

[Sin14b] SINITSYN, Valentine: *Understanding the Jailhouse hypervisor, part 2.* LWN.net. http://lwn.net/Articles/578852/. Version: January 2014 15, 22

[slo] http://www.dwheeler.com/sloccount/ 96, 127

[Spi00] SPINELLIS, Diomidis: Reflection as a mechanism for software integrity verification. In: *ACM Trans. Inf. Syst. Secur.* 3 (2000), Nr. 1, 51–62. http://dx.doi.org/10.1145/353323.353383. – DOI 10.1145/353323.353383 32

[STRE06] STUMPF, Frederic ; TAFRESCHI, Omid ; RÖDER, Patrick ; ECKERT, Claudia: A Robust Integrity Reporting Protocol for Remote Attestation. In: *Second Workshop on Advances in Trusted Computing (WATC '06 Fall).* Tokyo, Japan, November 2006 107, 108

[TB14] Tanenbaum, Andrew S. ; Bos, Herbert: *Modern Operating Systems (4th Edition)*. 4. Prentice Hall, 2014 `http://amazon.com/o/ASIN/013359162X/`. – ISBN 9780133591620 6

[tbo] `http://sourceforge.net/projects/tboot/` 53

[TCG05] TCG: TCG PC Client Specific TPM Interface Specification / Trusted Computing Group. 2005 (1.2). – Forschungsbericht 32, 34, 39, 56, 84, 102, 104, 109

[TCG07] TCG: TCG Specification Architecture Overview / Trusted Computing Group. 2007 (1.4). – Forschungsbericht 29, 30, 32, 33

[TCG11a] TCG: TPM Main Part 1 Design Principles / Trusted Computing Group. 2011 (116). – Forschungsbericht 32, 33, 104, 109

[TCG11b] TCG: TPM Main Part 2 TPM Structures / Trusted Computing Group. 2011 (116). – Forschungsbericht 32

[TCG11c] TCG: TPM Main Part 3 Commands / Trusted Computing Group. 2011 (116). – Forschungsbericht 32

[TCG12] TCG: TCG PC Client Specific Implementation Specification for Conventional BIOS / Trusted Computing Group. 2012 (1.21). – Forschungsbericht 29, 32, 37, 38, 39, 40, 105

[TCG13] TCG: TCG D-RTM Architecture / Trusted Computing Group. 2013 (1.0.0). – Forschungsbericht 37, 40

[VMw09] VMware: Performance Evaluation of Intel EPT Hardware Assist / VMware. 2009. – Forschungsbericht 7, 10

[WP10] White, Joshua ; Pilbeam, Adam: A Survey of Virtualization Technologies With Performance Testing. In: *CoRR* abs/1010.3233 (2010), 0. `http://arxiv.org/abs/1010.3233` 1, 15

[WR09] Wojtczuk, Rafal ; Rutkowska, Joanna: Attacking intel trusted execution technology / Invisible Things Lab. 2009. – Forschungsbericht 115

[WRC08] Willmann, Paul ; Rixner, Scott ; Cox, Alan L.: Protection Strategies for Direct Access to Virtualized I/O Devices. In: *USENIX 2008 Annual Technical Conference on Annual Technical Conference*. Berkeley, CA, USA : USENIX Association, 2008 (ATC'08), 15–28 12

[WSC$^+$07] Willmann, P. ; Shafer, J. ; Carr, D. ; Rixner, S. ; Cox, A.L. ; Zwaenepoel, W. ; Zwaenepoel, W.: Concurrent Direct Network Access for Virtual Machine Monitors. In: *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, 2007, S. 306–317 12

[XLG$^+$13] Xi, Sisu ; Lu, Chenyang ; Gill, Christopher ; Xu, Meng ; Phan, Linh T. ; Lee, Insup ; Sokolsky, Oleg: Global Real-Time Multi-Core Virtual Machine Scheduling in Xen / Washington University. 2013. – Forschungsbericht 15