Modeling and Abstraction of Memory Management in a Hypervisor

Pauline Bolignano[1,2(✉)], Thomas Jensen[1], and Vincent Siles[2]

[1] Inria, Rennes, France
{pauline.bolignano,thomas.jensen}@inria.fr
[2] Prove & Run, 77 Avenue Niel, 75017 Paris, France
{pauline.bolignano,vincent.siles}@provenrun.com

Abstract. Hypervisors must isolate memories of guest operating systems. This paper is concerned with proving memory isolation properties about the virtualization of the memory management unit provided by a hypervisor through shadow page tables. We conduct the proofs using abstraction techniques between high-level and low-level descriptions of the system, based on techniques from previous work on formally proving memory isolation in micro-kernels. The present paper shows how a hypervisor developed by Technische Universität Berlin has been formalized and presents the isolation properties we have proved on the targeted abstract model. In particular, we provide details about how the management of page tables has been formally modeled.

1 Introduction

A hypervisor is a software that makes it possible to run several guest operating systems (OS) on the same hardware. It is responsible for enforcing isolation between guests, for supervising their communication, *etc.* Hypervisors usually run at a privileged level, where all instructions are executable, whereas only some instructions are available to the guests. Their key role in managing the resources of the hardware make them highly security-critical components.

An OS running on bare metal manages the piece of hardware responsible for the memory management, called the Memory Management Unit (MMU). On every memory access, the MMU translates the virtual addresses manipulated by the software into physical addresses. The mappings from virtual to physical addresses are kept in page tables (PT) and managed by the OS. However when an OS runs on top of a hypervisor, the latter is the one managing the MMU and the translations. The hypervisor emulates the MMU for the guest OS and supervises the translations by maintaining PTs that *shadow* the PTs of the guest OS, called the Shadow Page Tables (SPTs). The SPT algorithms control the access of the guests to memory resources, and are thus central when proving security properties of guest OSes. Yet, they are definitely non-trivial, and considered an important challenge in formal OS development [1,6,7].

The motivation for our approach is obtaining the certification of isolation properties between guest OSes according to criteria such as Common Criteria [8].

P. Stevens and A. Wąsowski (Eds.): FASE 2016, LNCS 9633, pp. 214–230, 2016.
DOI: 10.1007/978-3-662-49665-7_13

One key element of this methodology is designing an abstract model of the concrete target of certification, and proving properties on this abstraction. We first build a low-level model of the hypervisor in the form of a transition system which represents its behavior precisely. This model is abstracted into a simpler transition system, in which properties are simpler to express and prove. We have written our model and conducted our proof with the language and tools developed at Prove & Run. These tools have already shown to be efficient in proving this kind of systems [12]. It should be stressed though, that the work presented here is independent of the particular tool used for its mechanized formalization.

The central part of our approach is that we abstract the *paged* address space (the memory and a pointer to the SPT) into a *linear* memory address space. To do this, we first provide a low-level model of the hypervisor and prove a set of key invariants of this model that are needed to prove isolation. This low-level model is then abstracted into a high-level model with separated memory segments. We have designed the abstract model to be as small as possible while keeping enough expressiveness to state our isolation property. This property guarantees that some resources of the guests are isolated from other guests. It can be divided in two sub-properties, concerning integrity and confidentiality, respectively. The *integrity* property for one guest ensures that its resources are not modified by other guests, unless it has given the authorization to do so. The *confidentiality* for one guest ensures that executions of other guests do not depend on its resources, unless it has given the authorization to do so. By its structure, the abstract model has inherent properties that ensure isolation, e.g. each guest has its own memory segments, whereas in the concrete model the memory is an array of bytes possibly shared with all the guests. To link the two models, we prove invariants on the concrete model which show that the whole system can be divided into well-separated subsystems.

Section 2 introduces the concept of page tables and shadow data structures. In Sect. 3, we outline the concrete model of a paravirtualized ARM version of a hypervisor developed by the SecT team at TU Berlin [17]. We give a classification of the transitions regarding the effects they have on the global state. We present the proof of an invariant which is essential for isolation. In Sect. 4 we present our abstract model, which is novel and interesting because it allows to precisely observe the memory while avoiding the notion of PTs. We show how we abstract the concrete memory. We then present the properties of integrity and confidentiality that we proved, and which taken together guarantee the isolation of guest memories. Finally, in Sect. 5 we discuss related work.

2 Memory Management in Hypervisors

When memory is virtualized, each entity runs as if it had the whole memory for itself, while the underlying platform shares the memory between several entities. In a classic OS with MMU, the OS keeps and manages the translations from virtual pages to physical pages in PTs. In the case of hypervision, a level of translation is added. The hypervisor may either use a hardware virtualization extension (if available) or implement a virtualization mechanism in software. We cover the latter scenario.

The hypervisor on which we work uses the most common software solution, based on SPT. SPTs are maintained by the hypervisor and translate guest virtual addresses (GVA) to physical addresses (PA), as illustrated in Fig. 1. The hypervisor creates and manages them by combining the Guest Page Tables (GPT), which translate GVA into guest physical addresses (GPA), and the Host Page Tables (HPT), which translate host virtual addresses HVA to PA. To simplify the presentation, we here consider that GPA and HVA are equal. The algorithm of managing SPTs we are working on is similar to those governing the Translation Lookaside Buffer (TLB) [5, Chapter 19]. For example when a page fault occurs at GVA *gva*, the hypervisor is notified. It goes through the GPTs to find out if any HVA *hva* was mapped to *gva* in the GPTs. If there is one, it computes the physical address *pa* corresponding to *hva* and, provided the guest is allowed to access this part of the memory, it adds the mapping from *gva* to *pa* in the SPTs. If the *gva* was not present in the GPTs, the hypervisor injects the page fault into the guest, so that the guest can add the mapping to the GPTs. Then the execution faults again on *gva*, because it is not yet in the SPTs, and it brings us back to the first case. Similarly when the guest switches PTs, when there is a TLB invalidation, the hypervisor handles the trap and updates the SPTs. In the hypervisor on which we work, the HPTs are allocated during system initialization, and a contiguous segment of PA corresponds to a contiguous segment of HVA.

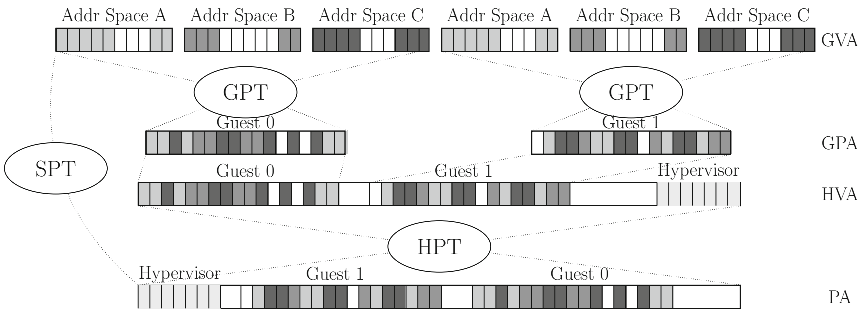

Fig. 1. Page tables of the hypervisor

3 The Concrete Hypervisor Model

The concrete model is the lowest level of our modeling effort; we kept it close to the implementation. However, unlike the C implementation, the effects of hypervisor and guest commands on memory are made explicit. We use untyped memory at this level to remain close to the C code. The closer the concrete model is to the C code, the smaller the gap is between what is proved and what is executed. Using untyped memory permits us to reason about type misinterpretation, and allows casts that are legitimate. We represent the memory as an array of bytes on which we use arithmetic computations to prove that there are no aliasing problems or overlapping data structures.

The memory that we model does not take the hypervisor memory space into account, except for the PTs about which we want to reason. Indeed, each time the hypervisor performs an action, it has an impact on its memory (e.g.pushing/popping something on its stack), and reasoning about these side-effects while reasoning about the effects of the action is not realistic. Moreover, we do not handle DMA and we do not model the devices' memory (cf Sect. 6).

3.1 Global State

We decompose a transition in three sub-transitions, the flow of execution is shown in Fig. 2. The hypervisor first restores the execution of the guest ($0 \to 1$), in particular, it sets the processor to *user mode*, which is an unprivileged mode, i.e.in the Privilege Level of execution 0 (PL0). Then the guest executes until it raises an exception or makes a hypercall, making the hardware switch to a privileged mode of execution ($1 \to 2$). A privileged mode is in the Privilege Level 1 (PL1). Depending on the level of privilege, some registers may or may not be visible, and accesses to registers may raise an exception. Finally the hypervisor saves the registers of the guest, then handles the call or the fault, while fixing the saved registers as necessary (step $2 \to 0$).

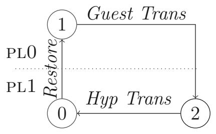

Fig. 2. Execution flow

The state $\sigma = \langle \sigma_{\text{HW}}, \sigma_{\text{HYP}}, \textit{exception} \rangle$ of the system is made of three components: the hardware state, the hypervisor state (which itself contains the states of the n guests) and an exception. The state of the hardware σ_{HW} is a tuple: $\langle \textit{mem}, \textit{base}, \textit{level}, \textit{regs}_{gp}, \textit{regs}_{mmu}, \textit{regs}_{gic} \rangle$.

Let *Addr* be the set of all the 32 bit addresses, *Byte* the set of all the bytes. The physical memory is a function from addresses to bytes: $\textit{mem} \in \textit{Mem} = \textit{Addr} \to \textit{Byte}$. In the hardware state σ_{HW}, the tuple $\textit{regs}_{\text{gp}}$ represents the values of the thirteen general purpose registers, the stack pointer, the link register and the program counter currently in the hardware, $\textit{regs}_{\text{gp}} = \langle r0, ..., r12, sp, lr, pc \rangle$. The *base* is the pointer to the root of the PT used by the processor for address translations. The tuple $\textit{regs}_{mmu}$ represents the fault status registers related to the MMU, which are needed to solve a page fault. The hardware state also provides the privilege level $\mathit{level} \in \{pl0, pl1\}$, and the registers of the Generic Interrupt Controller ($\mathit{regs}_{\text{gic}}$), which concern the management of the interrupts.

In a configuration with n guests, a guest is identified by an index in $\{1, ..., n\}$. The hypervisor state σ_{HYP} keeps the index of the current guest, its internal state, and the states of all the guests.

$$\begin{gathered}\sigma_{\text{HYP}} = \langle \mathit{curr}, \sigma_{\text{int}}, \langle \sigma_{G1}, ..., \sigma_{Gn} \rangle \rangle \\ \text{where, } \forall i, \sigma_{Gi} = \langle \mathit{vbase}, \mathit{vmode}, \mathit{vbnk}, \mathit{vregs}_{\text{gp}}, \mathit{vregs}_{\text{mmu}}, \mathit{vregs}_{\text{gic}} \rangle\end{gathered}$$

The emulated PT base pointer (*vbase*) of the guest contains a pointer to the GPT, i.e. from GVA to GPA. When a page fault occurs, the hypervisor uses the *vbase* to walk the GPT. However the hardware *base* pointer never contains the *vbase* pointer but rather the pointer to the SPT or the HPT.

3.2 Page Tables

A PT maps virtual addresses to physical addresses and provides the access rights to the address. The set of rights is denoted by *Rights* and contains two elements: $\{rw, ro\}$. We define the total order relation $\geq$ over *Rights* by $rw \geq ro$. The set of page tables PT is defined by $PT = \mathit{Addr} \rightarrow \mathit{Addr} \times \mathit{Rights}$, these functions are not total, a virtual address $va \in \mathit{Addr}$ not in the domain corresponds to an address for which there is no translation, i.e. the access to va would raise a fault. The function *pt* takes a memory and a pointer and returns the PT located there: $pt(\mathit{mem}, \mathit{base})$ is read as"the page table at address *base* in memory *mem*", $pt \in \mathit{Mem} \rightarrow \mathit{Addr} \rightarrow PT$. We denote by Γ_f the graph of function f, in particular $\Gamma_{pt(\mathit{mem},\mathit{base})}$ is the set of mappings present in the PT at address *base* in memory *mem*. In practice, when looking for the physical address corresponding to the virtual address va, va is split in three parts: the first part is an index i_1 in the first level of PT, the second is an index i_2 in a second level of PT, and the last is an offset in a page. The *base* address in $pt(\mathit{mem}, \mathit{base})$ is the base address of the first level of PT, each entry of the first level of PT either holds a fault or the address to a second level of PTs. Similarly to *pt*, we note pt_2 the function that takes a memory and a base address and returns a second level of PT.

An address which is not in the image of a PT is *not mapped*, whereas an address in the image is *mapped* with some rights. For a page table $\mathit{table} \in PT$, we denote the first projection of $\mathit{Im}(\mathit{table})$ by $\mathit{Map}(\mathit{table})$. Similarly, we use $\mathit{Map}_{\text{RW}}(\mathit{table})$ to denote the set of all the physical addresses mapped with RW rights by *table*: $\mathit{Map}_{\text{RW}}(\mathit{table}) = \{pa | (pa, rw) \in \mathit{Im}(\mathit{table})\}$. The hypervisor associates a SPT to each GPT, we note $\mathcal{B}_{\text{SPT}}(\sigma_{\text{int}}, i)$ the set of base addresses of the SPTs of guest i.

We define m non-overlapping intervals $I_1, ..., I_m$ of physical addresses, such that $\bigcup_{k=1}^{m} I_k \subset \mathit{Dom}(\mathit{mem})$. We let I represent the set $\{I_1, ..., I_m\}$. During the execution, the hypervisor ensures that the addresses of each interval are only mapped in the SPT of the allowed guests. The permissions for each interval are provided by the initial configuration through the *region* function. The function *region* takes an interval and a guest index and returns the maximum rights that the guest can have on this interval: $region \in (I \times \{1,..,n\}) \rightarrow Rights$. The function is partial, $(I_j, i) \notin Dom(region)$ means that the guest i has no rights on the interval I_j. The hypervisor ensures that an address in an interval is always mapped in the SPT of a guest with rights inferior or equal to the rights defined by the *region* function (see Invariant 1 in Sect. 3.4). The relation $allowed(a, i, r)$ is true if the address a is in a region of guest i with rights r: $\exists k, a \in I_k \wedge region(I_k, i) = r$. An interval might be private or shared between two guests. If shared, one guest has RO access and the other RW access to it, i.e. it is a one way buffer. Let j be in $\{1, ..., m\}$. Let i and k be in $\{1, ..., n\}$. The following predicates formalize the two possible configurations of an interval:

- $private(I_j, i) \Leftrightarrow \forall a \in I_j, \forall l \neq i, allowed(a, i, rw) \wedge \neg allowed(a, l, \_)$
- $shared(I_j, i, k) \Leftrightarrow \forall a \in I_j, (allowed(a, i, rw) \wedge allowed(a, k, ro) \wedge \forall l \notin \{i, k\}, \neg allowed(a, l, \_))$

3.3 Concrete Transitions

A transition of the system is decomposed into three transitions, as defined in Fig. 2. The *restore transition* models the change from privileged mode to user mode. The hypervisor injects an interrupt to the guest beforehand if any is pending. We define the two other types of sub-transitions below.

Guest transitions occur in user mode. We confine the possible effects they can have on the system by making two hypotheses on the processor. First, the guest may only change the non-privileged registers $regs_{\mathrm{gp}}, regs_{\mathrm{mmu}}$ and $regs_{\mathrm{gic}}$. In particular, it *cannot* change the PT base register. Secondly, it may only change the memory mapped in RW by the PT currently used by the hardware ($pt(mem, base)$). This second hypothesis only makes sense if the so-called current PT is constant during a guest transition. This is ensured by an invariant stating that the memory space where SPT are stored is not mapped in RW by any guest. We denote by $wf(mem, base)$ the property stating that the PT at address *base* does not map itself in the memory *mem*. The guest transition is depicted in Fig. 3, where the notation $mem' \cong mem[map_{RW}(pt(mem, base))]$ means that *mem* and mem' are equal except at the physical addresses mapped in RW by the PT at address *base* in *mem*. For readability, fields modified by the transition are represented bolded.

As previously explained, the guest execution gives back the control to the hypervisor when it raises an exception. That is why when the guest ends, the field exception e is always updated. In the case of an abort, information about the fault is stored in the $regs_{\mathrm{mmu}}$. As stated, the guest transition does not capture a third hypothesis on the processor, which is fundamental to prove confidentiality. Thus, in Axiom 1 we state that a guest transition only depends on the part of the memory mapped by the current PTs. Here, $\sigma \overset{A}{=} \sigma'$ means that the restriction to A of the memories of two states σ and σ' are equal and that all their other fields are equal.

Axiom 1. Let σ_1 and σ_2 be two states such that $\sigma_1 \overset{A}{=} \sigma_2$, where $A = map(pt(\sigma_1.base, \sigma_1.mem))$. If $\sigma_1 \xrightarrow{GuestTrans} \sigma_1'$ then $\sigma_2 \xrightarrow{GuestTrans} \sigma_2'$ and $\sigma_1' \overset{A}{=} \sigma_2'$.

GUEST TRANS:

$$\frac{\mathit{wf}(\mathit{mem}, \mathit{base}) \qquad \mathit{mem}' \cong \mathit{mem}[\mathit{map}_{RW}(\mathit{pt}(\mathit{mem}, \mathit{base}))]}{\left\langle \begin{array}{l} \langle \boldsymbol{mem}, \mathit{base}, \boldsymbol{pl0}, \boldsymbol{regs}_{\mathbf{gp}}, \boldsymbol{regs}_{\mathbf{mmu}}, \boldsymbol{regs}_{\mathbf{gic}} \rangle \\ \sigma_{\mathrm{HYP}} \\ \boldsymbol{e} \end{array} \right\rangle \rightarrow \left\langle \begin{array}{l} \langle \boldsymbol{mem'}, \mathit{base}, \boldsymbol{pl1}, \boldsymbol{regs'}_{\mathbf{gp}}, \boldsymbol{regs'}_{\mathbf{mmu}}, \boldsymbol{regs'}_{\mathbf{gic}} \rangle \\ \sigma_{\mathrm{HYP}} \\ \boldsymbol{e'} \end{array} \right\rangle}$$

PG FAULT:

$$\frac{\begin{array}{c} \mathit{decode}(\mathit{abt}, \sigma_{\mathrm{HW}}) = \mathit{pf}(\mathit{gva}) \\ \sigma_{Gi}.\mathit{vregs}_{\mathrm{mmu}}.\mathit{pg} = \mathit{enabled} \qquad \mathit{hpt}(\sigma_{Gi}.\mathit{vbase}) = (\mathit{pbase}, \_) \\ \{(\mathit{gva}, (\mathit{gpa}, r_0))\} \in \Gamma_{\mathit{pt}(\mathit{mem},\mathit{pbase})} \qquad \mathit{hpt}(\mathit{gpa}) = (\mathit{pa}, \_) \\ \exists r_1 \geq r_0, \mathit{allowed}(\mathit{pa}, i, r_1) \qquad \Gamma_{\mathit{pt}(\mathit{mem}',\mathit{base})} = \Gamma_{\mathit{pt}(\mathit{mem},\mathit{base})} \cup \{(\mathit{gva}, (\mathit{pa}, r_0))\} \\ \sigma'_{\mathrm{HYP}} = \sigma_{\mathrm{HYP}}[\sigma_{\mathrm{int}} \leftarrow \mathit{alloc}(\sigma_{\mathrm{int}}, \mathit{mem}, \mathit{gva})] \end{array}}{\left\langle \begin{array}{l} \langle \boldsymbol{mem}, \mathit{base}, \mathit{regs}_{\mathrm{mmu}}, \mathit{regs}_{\mathrm{gp}}, \mathit{pl1}, \mathit{regs}_{\mathrm{gic}} \rangle \\ \boldsymbol{\sigma}_{\mathbf{HYP}} \\ \mathit{abt} \end{array} \right\rangle \rightarrow \left\langle \begin{array}{l} \langle \boldsymbol{mem'}, \mathit{base}, \mathit{regs}_{\mathrm{mmu}}, \mathit{regs}_{\mathrm{gp}}, \mathit{pl1}, \mathit{regs}_{\mathrm{gic}} \rangle \\ \boldsymbol{\sigma'}_{\mathbf{HYP}} \\ \mathit{abt} \end{array} \right\rangle}$$

Fig. 3. Concrete guest transition and page fault transition

Hypervisor transitions happen in a privileged mode. There are fourteen hypervisor transitions, that can be grouped as follows. The first group contains the six transitions related to the memory management: they either modify the current SPTs or the base pointer. The second group only contains the scheduling transition. It corresponds to the guest context switch, that loads the registers of the new guest. In particular, it changes the PT base pointer. The third group contains three transitions that inject a fault to the guest. These transitions only have an impact on general purpose registers. The fourth group contains the emulation of the access to privileged registers. The hypervisor writes to the corresponding emulated privileged register, or reads its value and put it in one of the general purpose registers. This transition may have an impact on the general purpose register or on the guest's GIC. Finally, the transitions of the fifth group concern the IRQs, they have an impact on the GIC registers of the hypervisor or of the guests.

As an example, a transition corresponding to a page fault is presented in Fig. 3. A page fault occurs when the guest tries to access an address *gva* which is not mapped in the current SPT, the first premise illustrates the decoding of the fault by the hypervisor. The second premise indicates that the MMU of the guest is activated. The GPA *vbase* is the base of the GPT that the guest is currently using, *pbase* is the corresponding physical address. The faulting address *gva* is in the domain of the GPT, and is bound to *pa* with the rights r_0. The physical address *pa* corresponds to *gpa* in the HPT, guest i is allowed to map it with r_0 rights. The memory mem' of the resulting state is such that the graph of the current SPT in mem' contains the new mapping $(\mathit{gva}, (\mathit{pa}, r_0))$. The internal state σ_{int} of the hypervisor might be modified if the addition of a new mapping necessitates the allocation of a new PT (it changes the state of the allocator), we do not detail the behavior of the *alloc* function here.

3.4 SPT Invariants

The main invariant needed in the concrete model is that if a physical address *pa* is mapped by one of the SPTs of a guest i with some rights r_0, then *pa* is in one of the intervals to which this guest has access, with compatible rights. We express this invariant formally, using the notations introduced in Sect. 3.2:

Invariant 1. $(base \in \mathcal{B}_{\text{SPT}}(\sigma_{\text{int}}, i) \wedge (pa, r_0) \in Im(pt(mem, base))) \Rightarrow \exists r_1 \geq r_0 \wedge allowed(pa, i, r_1)$

We have seen in Sect. 3.3 that the page fault handling may lead to the addition of a new mapping in the SPT of the guest. The addition and removal of mappings in a SPT are the crucial parts of the algorithm when it comes to isolation. We have proved the preservation of Invariant 1 during this sensitive operation. We present below the main invariants needed for the proof.

We denote by $Pool(i)$ the set of physical addresses where the SPT of the guest i might be located. The static configuration ensures that the physical addresses of the pools are not in a part of memory attributed to a guest: $\forall i, \forall pa \in Pool(i) \Rightarrow \forall j, r, \neg allowed(pa, j, r)$. We say that $part\_of(mem, b, a)$ is true if the byte at address a holds any value of the PT at base address b.

Invariant 2 ensures that the SPTs of a guest are located within its pool. For each guest, the hypervisor references the free slots available to allocate a new SPT, we write $free\_pt_2(\sigma_{\text{int}}, j, b)$ if b is the address of a free slot for guest j. Invariant 3 states that the free slots for a guest are in its pool, it allows to prove that the former invariant holds after the allocation of a new level of SPT.

Invariant 2 (SPTs disjoint Pools). $b \in \mathcal{B}_{\text{SPT}}(\sigma_{\text{int}}, i) \wedge part\_of(mem, b, pa) \Rightarrow pa \in Pool(i)$.

Invariant 3 (Free PTs disjoint Pools). $free\_pt_2(\sigma_{\text{int}}, j, b_2) \Rightarrow b_2 \in Pool(j)$.

In order to maintain Invariant 1 during the allocation of a new PT in the SPT of guest i, we need to know that the new PT does not map any physical address outside the range allowed to guest i. This property can be stated more easily if a new PT is flushed before being attributed, yet Invariant 4 is sufficient in order to ensure isolation.

Invariant 4 (Free PT allowed). $free\_pt_2(\sigma_{\text{int}}, j, b_2) \wedge (pa, r_a) \in Im(pt_2(mem, b_2)) \Rightarrow (\exists r_b \geq r_a \wedge allowed(pa, j, r_b))$.

Last but not least, SPTs must not overlap, i.e. addresses where a part of a SPT is kept must not correspond to addresses where another part of any SPT is kept. We write $overlap(mem, b, b')$ if the SPT at base b in memory mem overlaps with the SPT at base b'. In particular $overlap(mem, b, b)$ means that the different branches of the same PT overlap. In order to prove that this invariant holds after the allocation of a new second level PT, we must also ensure that a free PT was not allocated in another SPT of that guest beforehand. Note that Invariant 2 already ensures that the PT was not allocated to another guest.

Invariant 5 (No overlap). $b, b' \in \mathcal{B}_{\text{SPT}}(\sigma_{\text{int}}, j) \Rightarrow \neg overlap(mem, b, b')$.

Invariant 6 (Free PT not allocated). $free\_pt_2(\sigma_{\text{int}}, j, b_2) \Rightarrow \forall b \in \mathcal{B}_{\text{SPT}}(\sigma_{\text{int}}, j), \neg part\_of(mem, b, b_2)$.

Preservation of Invariant 1: Consider the case where the hypervisor adds a new mapping m_{new} from the GVA va to the physical address pa in a SPT of guest j, with rights r such that $allowed(pa, j, r)$. We write $base$ for the physical base address of the SPT, and let i_1 and i_2 be the indexes in the first and second level PTs for va. The hypervisor evaluates the i_1^{th} descriptor of the first level of PT. Suppose that this descriptor is a fault. The hypervisor performs three steps: (1) it searches for a free second level PT in the pool of guest j, marks it as used and returns its base address $base_2$, (2) it modifies the first level descriptor at the i_1^{th} entry in the first level PT at base $base$ so that it points to $base_2$, (3) it modifies the second level descriptor at the i_2^{th} entry in the second level of PT freshly allocated so that it leads to pa with the rights r.

We sketch the proof of preservation of Invariant 1. Let mem (resp. mem') be the memory before (resp. after) the transition. We assume that Invariant 1 holds for mem (H_{inv}). Assume that there exists a mapping $m_{\text{bad}}{=}(va', (pa', r'))$ which contradicts Invariant 1 in mem' (H_{break}). Let $base_k$ be the base of a SPT of a guest k in which the mapping m_{bad} is: $m_{\text{bad}} \in \Gamma_{pt(mem', basek)}$. We write i'_1 and i'_2 for the decomposition of the virtual address va' into indexes in the first and second level PT. We proceed by analyzing all the possible branches in all the SPTs where m_{bad} can be, and for each case we refute the existence of such a mapping. The following lines summarize the hypotheses:

- $m_{\text{new}} = (va, (pa, r))$ is the new mapping added in the SPT of guest j, located at base $base_j$.
- $base_2$ is the base of the second level of PT freshly allocated.
- H_{pa}: The mapping to be inserted is allowed ($allowed(pa, j, r)$).
- H_{inv}: Invariant 1 holds for mem.
- $m_{\text{bad}} = (va', (pa', r'))$.
- H_{break}: Invariant 1 does not holds for mem ($m_{\text{bad}} \in \Gamma_{pt(mem', basek)} \wedge \nexists r_0 \geq r', allowed(pa', j, r_0)$).
- va can be decomposed in i_1 and i_2.
- va' can be decomposed in i'_1 and i'_2.

Proof. Case $k = j$, $base_k = base_j$ and $i'_1 = i_1$: this case means that m_{bad} is one of the mapping that we have just added, i.e. that the address pa' is mapped by the second level of PT we have just allocated.

◇ Case $i'_2 = i_2$: it means that m is the very mapping we have added, $m = (va, (pa, r))$[1]. Yet we know that this mapping is allowed for guest j ($allowed(pa, j, r)$), which contradicts H_{break}.

◇ Case $i'_2 \neq i_2$: it means that the address pa' is mapped by the second level PT we have just added but does not correspond to the page at index i_2. From Invariant 4 we know that all the indexes of the new PT are in an allowed range for the guest, so it contradicts H_{break}.

[1] In reality we just know that m is in the page that we have just mapped, other invariants and conditions on the arguments must be verified (e.g. that the addresses va and pa are aligned to the size of a page) but we do not introduce all the details here.

Case $k \neq j$, *or* $base_k \neq base_j$ *or* $i'_1 \neq i_1$: these are the cases where m is in another branch than the modified one, we show that it was necessarily present before, thus contradicting H_{inv}.

- ♢ Case $k \neq j$: From Invariants 2 and 3, we know that $pt(mem, base_k) = pt(mem', base_k)$. Thus H_{break} yields that m_{bad} was already present in mem ($m_{\text{bad}} \in \Gamma_{pt(mem,basek)}$), which contradicts H_{inv}.
- ♢ Case $k = j \wedge base_k \neq base_j$: Invariants 6 and 5 lead to the same conclusion as the precedent case.
- ♢ Case $k = j \wedge base_k = base_j \wedge i'_1 \neq i_1$: From Invariants 6 and 5 we know that the change we made in one branch i_1 of the SPT does not affects other branches, in particular the one in i'_1. Thus m_{bad} was necessarily in the SPT before the addition of the mapping, contradicting H_{inv}.

Note that the hypervisor itself is virtualized by the HPT (i.e. it manipulates virtual addresses which are translated by the processor with the HPT). The addition of a new mapping in the SPT that we have described hides that when the hypervisor accesses the i^{th} entry of a PT located at some physical address, it refers to the entry by its virtual address. Hence one must ensure that the traversal of the SPTs made by the hypervisor with *virtual* addresses is equivalent as the one that would be made with *physical* addresses. We do not present the invariants here, yet it must be underlined that these invariants are used in each case of the proof to specify the effects of the actions of the hypervisor.

The invariants presented are not tight to implementation, they concern SPT algorithms in general. We do have some properties for free due to our particular static configuration, e.g. we know that the pools and the guest regions are disjoint. In a dynamic configuration, this kind of properties would have to be proved, but the reasoning stays unchanged.

The preservation of Invariant 1 under the addition of a mapping is the major requirement to formally link the page fault transition that we have presented in Fig. 3 to the abstract transition *mm* that we present later in Sect. 4.2.

4 The Abstract Hypervisor Model

This section presents the abstract model used to prove that SPTs provide memory isolation between guests. Some data structures and algorithms of the concrete model have no impact on the isolation property. Thus, provided the right invariants are proved on the concrete model, there is no need to project them on the abstract state. For example, in our case the generic interruption controller has no effect on the memory management; therefore we can remove it from the abstract model and remove all the derived operations. That is why the abstract state is much smaller.

The abstract state contains the index of the current guest and the states of all the guests:

$$\sigma_\alpha = \langle curr, \sigma_1, ...\sigma_n \rangle, \text{ where } \sigma_i = \langle abs\_regs, priv, \langle s_1, ..., s_n \rangle, \langle r_1, ..., r_n \rangle \rangle$$

The guest state is composed of some abstract registers and an address space. We model the address space as a set of segments. Each guest has: (1) a private segment, (2) n shared segments to which it has *write* access, called the *send* segments, (3) n shared segments to which it has *read* access, called the *receive* segments. A segment has the type $Addr \rightarrow Cell$, where $Cell$ represents a byte either mapped or not: $Cell = Byte \times Bool$[2]. We say that two cells have the same value if they have the same byte value. We say that two cells have the same tag if they have the same boolean value. The segment function is not total. The segment s_j of σ_i represents the segment in which i can write and j can read. The segment r_j of σ_i represents the segment in which i can read and j can write. The segments are duplicated, such that the j^{th} send segment of guest i is synchronized with the i^{th} receive segment of guest j. Two segments are said to be synchronized if they have the same values but possibly different tags. The notion of synchronization instead of a mere equality allows to reason about sharing, by capturing the fact that a byte value at some address in some segment of a guest can change even if the guest does not map the address. Such an abstraction allows to have a precise view of the memory while discarding the PTs.

4.1 Link Between the Concrete and the Abstract Model

In this section we show how the concrete and the abstract models can be linked.

For a guest i, the abstract cell corresponding to the byte b at address pa in memory mem is such that (1) the value of the cell is b (2) the tag of the cell is mapped if pa is mapped by the current SPT of guest i, unmapped otherwise.

However not all the memory addresses of the concrete model are to be abstracted in the segments of a guest i, we define below which addresses are abstracted for each guest, and in which segment they are located. Recall from the concrete guest transition that when a guest runs, it only has access to the addresses mapped by the PT currently in the base register of the processor. We assume in the sequel that when a guest runs, only one of its own SPT can be used by the processor (the preservation of this invariant is obvious).

All the addresses which are in the domain of a segment of the abstract guest i correspond to all the physical addresses that the concrete guest i can possibly access. Thus it corresponds to all the addresses that the guest i might map in its SPTs, formally it corresponds to the set:

$$\{pa \in Addr | \exists r_0, \exists base \in \mathcal{B}_{\mathrm{SPT}}(\sigma_{\mathrm{int}}, i) \wedge (pa, r_0) \in Im(pt(mem, base))\}$$

When Invariant 1 is verified, we can characterize this set by the addresses located in the intervals on which guest i has some rights, that is the set:

$$\{pa \in Addr | \exists r_1, allowed(pa, i, r_1)\}$$

Thus, we can bound the domains of the segments to the intervals defined in Sect. 3.2. The fact that this part of the abstraction does not depend on the SPTs

[2] In fact we differentiate a byte mapped in RW and a byte mapped in RO, but we omit the details here for clarity's sake.

but rather on the definition of intervals is convenient. Indeed it simplifies the proof of correspondence between an abstract and a concrete action, particularly if the action has an impact on the SPTs.

Now that we have a characterization of all the addresses that are in the segments of a guest, we dispatch them in the segments with the right properties. For example if an address is in an interval over which guest i has RW access and guest k has RO access, it appears in the k^{th} send segment of guest i and in the i^{th} receive segment of guest k. Recall from Sect. 3.2 that an interval can be in two configurations, shared or private. The correspondence between the concrete intervals and the segments is defined as follow:

- $pa \in I_j \wedge private(I_j, i) \Leftrightarrow pa \in Dom(\sigma_i.priv)$
- $pa \in I_j \wedge shared(I_j, i, k) \Leftrightarrow pa \in Dom(\sigma_i.s_k) \wedge pa \in Dom(\sigma_k.r_i)$

4.2 Abstract Transitions

We present here the abstractions of the transitions presented in Sect. 3.3, i.e. the *restore*, *guest*, and *hypervisor transitions*.

The view of the concrete **restore transition** does either nothing (in case just the PL is changed) or injects an IRQ into the guest, which only impacts the registers.

The whole **guest transition** is represented in Fig. 4, but in practice, we split the guest transition in two steps. The first part, called the *abstract run*, is the view of the concrete guest transition seen from the current guest. Invariants on the concrete level allow to state that only the writable segments of the abstract guest (private and send segments) are modified during the run, formally: $\sigma' = run(\sigma) \Rightarrow \sigma' \cong \sigma[r_1...r_n]$. Secondly, changes in the send segments of the current guest are mirrored in the corresponding receive segments of the other guests. In other terms, the receive segments of the other guests are synchronized with the send segment of the current guest. The synchronization seg'_1 of segment seg_1 with seg_2, i.e. updating all the values of seg_1 with those of seg_2 without changing its tags, is denoted by $seg'_1 = seg_1 \xleftarrow{VAL} seg_2$.

We distinguish four types of **hypervisor transitions**, depending on their impact on the guest states. Each transition corresponds to one or several groups of the hypervisor concrete transitions presented in Sect. 3.3. The *change register* (*chreg*) transition is the abstraction of the concrete injection transitions and of the access to privileged register transition. The particularity of these transitions is that their only observable impact is on registers. The *memory management* (*mm*) transition captures the effects of the concrete transitions concerning the memory management virtualization. These concrete transitions have an impact on the registers and on memory. In particular, they do not change the value of memory cells but only the active SPT. It means that the impact of the *mm* transition on the segments is only on their tag (i.e. mapped or not). The *nop* transition is the abstraction of all the concrete transitions which do not have any observable impact on the abstract state. It abstracts all the IRQ transitions, indeed, all these transitions only impact the GIC which we do not represent in the abstract model. The *scheduling* (*sched*) transition corresponds to the concrete scheduling transition.

$$\frac{\sigma_i' = run(\sigma_i) \qquad \forall k \neq i, \sigma_k' = \sigma_k[r_i \xleftarrow{VAL} \sigma_i'.s_k]}{\langle i, \boldsymbol{\sigma_1}, ...\boldsymbol{\sigma_n}\rangle \rightarrow \langle i, \boldsymbol{\sigma_1'}, ...\boldsymbol{\sigma_n'}\rangle}\ \textsc{guest trans}$$

$$\frac{decode(\sigma_i.abs\_regs) = inject(r') \qquad \sigma_i' = \sigma_i[abs\_regs \leftarrow r']}{\langle i, \sigma_1, ..., \boldsymbol{\sigma_i}, ...\sigma_n\rangle \rightarrow \langle i, \sigma_1, ..., \boldsymbol{\sigma_i'}, ...\sigma_n\rangle}\ \textsc{chreg}$$

$$\frac{decode(\sigma_i.abs\_regs) = sched(nxt)}{\langle \boldsymbol{i}, \sigma_1, ...\sigma_n\rangle \rightarrow \langle \boldsymbol{nxt}, \sigma_1, ...\sigma_n\rangle}\ \textsc{sched}$$

$$\frac{decode(regs) = nop}{\langle i, \sigma_1, ...\sigma_n\rangle\,. \rightarrow \langle i, \sigma_1, ...\sigma_n\rangle}\ \textsc{nop}$$

$$\frac{decode(\sigma_i.abs\_regs) = mm(\sigma_i')}{\langle i, \sigma_1, ...\boldsymbol{\sigma_i}, ...\sigma_n\rangle \rightarrow \langle i, \sigma_1, ...\boldsymbol{\sigma_i'}, ...\sigma_n\rangle}\ \textsc{mm}$$

$$\frac{\sigma_i \cong \sigma_i'[abs\_regs]}{\langle i, \sigma_1, ...\boldsymbol{\sigma_i}, ...\sigma_n\rangle \rightarrow \langle i, \sigma_1, ...\boldsymbol{\sigma_i'}, ...\sigma_n\rangle}\ \textsc{restore}$$

Fig. 4. Abstract sub-transitions

In order to establish a formal link between the two models, we need to prove the correspondence between all the transitions of the models. More specifically, we need to prove that if a concrete and an abstract state are related by the abstract relation defined in Sect. 4.1, the two states resulting from two corresponding transitions are also related. The correspondence proofs rely on the proof of preservation of invariants presented in Sect. 3.4. The preservation of the low-level invariants is the most difficult part of the proof. Therefore the correspondence of the transitions related to memory management are the most subtle to prove, because they may modify the SPT, making the preservation of the invariant more difficult to ensure. We have already proved the preservation of our invariants under the map operation which is used in the page fault transition, in order to prove its correspondence with the abstract *mm* transition. We have completed the proof of correspondence between the abstract and the concrete guest transitions. We have thus partially validated our abstraction.

4.3 Properties

Guests may interfere with each other (e.g.through shared memory), so we cannot prove non-interference. Instead we prove an *isolation* property on some resources of the guests, i.e. we prove their integrity and their confidentiality. The resources on which we prove isolation are the registers and the memory segments. Below, we detail the properties on segments, as our main focus is the memory isolation. We express the properties on one transition step. More exactly, to state a property for one guest, we confine the effects that the execution of another guest can have on the former. Thus, as our system is sequential, we consider a transition where the former guest does not run. We prove the extension of these properties to any sequence of transitions where a guest does not run. The proof sketch of integrity shows the simplicity with which we can bound the effects of a transition in our model. The proof of confidentiality is done in a similar way.

Integrity for a guest i means that if another guest j runs, then the private segment and the send segments of i are not modified, and only its j^{th} receive segment might have change.

Theorem 1 (Integrity). *Let i and j be two guest indexes such that $i \neq j$. Consider a transition where j is the running guest. If*

$$\langle j, \sigma_1, ..., \sigma_i, ..., \sigma_n \rangle \to \langle j', \sigma'_1, ..., \sigma'_i, ..., \sigma'_n \rangle$$

then $\sigma'_i.priv = \sigma_i.priv$ *and* $\forall k, \sigma'_i.s_k = \sigma_i.s_k$ *and* $\forall k \neq j, \sigma'_i.r_k = \sigma_i.r_k$.

Proof. Let i and j be two guest indexes such that $i \neq j$. We consider a transition from the state $\langle j, \sigma_0, ..., \sigma_n \rangle$. The first part of the transition is the restore transition, which does not change the running guest nor the state of guest i. The second part of the transition is the guest transition. From the definition of the guest transition (Fig. 4) we know that the running guest is not changed, and that the state σ'_i of guest i after the transition is such that: $\sigma'_i = \sigma_i[r_j \xleftarrow{VAL} \sigma'_j.s_i]$. Hence the three following facts are verified: $\sigma'_i.priv = \sigma_i.priv$ and $\forall k, \sigma'_i.s_{ik} = \sigma_i.s_k$ and $\forall k \neq j, \sigma'_i.r_k = \sigma_i.r_k$. The third part of the transition is the hypervisor transition. None of the four hypervisor transitions changes the state of guest i. Therefore integrity is verified for any transition.

To express confidentiality properties, we compare one step of execution from two states which differ only on some resources of guest i. If the guest j which runs in this step has no authorization to access these resources, then the two states resulting from the transition are equal except on guest i. Notice that we cannot prove our property on a non-deterministic system because we reason on the very fact that two executions end in similar states. Yet the scheduling and restore sub-transitions are non-deterministic, indeed we have not included in our model sufficient information to decide whether an interrupt is to be injected or which guest is to be run on a scheduling. We address this issue by making two assumptions that allow us to add some extra information which make these sub-transitions deterministic. We suppose that when a guest does not run, its memory does not interfere with the scheduler nor with the interrupt management of other guests. There is no major difficulty in proving these properties but we set them aside in the first instance in order to focus on memory isolation. Therefore, we reason with two extra arguments which make the system deterministic: a guest to be run next (nxt) and the optional registers corresponding to the IRQ injection ($oirq$). We write $\xrightarrow{nxt,oirq}$ for such an enhanced transition.

Theorem 2 (Confidentiality). *Let i, k and j be three guest indexes such that $i \neq j$ and $k \neq j$. Let $\hat{\sigma_i}$ be a guest state such that $\hat{\sigma_i} \cong \sigma_i[priv, s_k, r_k]$. If*

$$\langle j, \sigma_1, ...\sigma_i, ..., \sigma_n \rangle \xrightarrow{nxt,oirq} \langle j', \sigma'_1, ..., \sigma'_i, ..., \sigma'_n \rangle$$

then

$$\langle j, \sigma_1, ...\hat{\sigma}_i, ..., \sigma_n \rangle \xrightarrow{nxt,oirq} \langle j', \sigma'_1, ...\hat{\sigma}_i{}', ..., \sigma'_n \rangle.$$

5 Related Work

Daum et al. [10] strengthened the refinement between the abstraction layers of the micro-kernel seL4 to reason about virtual memory management. It allows them to reason at a finer granularity and to have an abstract model upon which they can extend their previous proofs [14] to prove isolation between processes. Although OSes and hypervisors have much in common, the memory management part is quite different, since the SPTs are not present in OSes.

Barthe et al. formalized in Coq an idealized model of a paravirtualized hypervisor [6]. They included the caches in their model and considered cache-based side-channel attacks, which is out of our scope. They do not refine their model to an implementation level, and they make several simplifications, such as considering only one level of page tables. In addition they do not consider any sharing between guests.

Blanchard et al. present a case study on the creation of a new mapping in a page table [7]. Their method is quite different from ours. They work on a part (i.e. one function) independently of the rest of the system whereas we model the interactions between the several parts of the system, to prove high-level properties on the whole system. In contrast to us, they consider parallelism and show that their model is valid for weak memory models.

In [1,11], Kovalev et al. present the proof of a correctness property of the TLB virtualization code, using the verifier VCC [9]. They prove that if a translation is present in the virtual TLB (i.e. the TLB that the guest would have if it were running directly on the hardware), it is also present, modulo some translation stages, in the hardware TLB (i.e. the cache of the SPT). Their property does not provide isolation, it is complementary to ours. Their hardware model is very complete and detailed, but their SPT algorithm is rather simplified. For example, they suppose that there is always a free SPT slot available when allocating a new one, whereas we go in deeper details in the model of the SPT allocator, as we consider that the proof of its well-formedness is a key aspect of the isolation proof.

Nemati et al. [15] prove isolation properties on a hypervisor which uses direct paging. Direct paging does not use shadow data structures. Though still supervised by the hypervisor, the guest OS directly manages mappings from GVA to PA. This solution requires additional modifications of the guest.

On a hypervisor supporting one guest, Vasudevan et al. [16] proved that the guest cannot write in hypervisor memory, i.e. they proved the integrity of the hypervisor memory. They verified automatically some modules of the hypervisor, using the CBMC model checker, and other manually, due to the limitation of the tool. Andrabi extended the automatic verification by proving the well-formedness of the PT setup in [3]. They do not virtualize the memory with SPTs, but rather use the hardware virtualization solution.

6 Conclusion

The management of page tables (PT) is a core task for a hypervisor and involves non-trivial algorithms which make it difficult to prove -and even to state- that the hypervisor enforces isolation between its guest OSes. In this paper we have argued that it is possible to construct an abstract model of a hypervisor, on which it is considerably simpler to conduct such proof. To this end, we have presented a concrete model of a hypervisor (in which six out of the fourteen hypervisor transitions concern the PTs), and established a number of invariants on this model. Based on these invariants, it is possible to construct an abstract model in which the management of PTs has been abstracted away. We have proved isolation on the resulting abstract model.

Handling page faults that lead to the addition of a mapping in the SPTs, is the most complex and security-critical operation of the SPT algorithm. Complex, because it modifies the structure of the SPT, requiring a substantial number of well-formedness invariants to capture the effects of the modifications. Security-critical, because it gives guests access to new parts of the memory. A central part of the work reported here has been to state and prove the low-level invariants needed to prove the correspondence of the concrete page fault operation with its abstract counterpart. The major part of the other operations of the SPT algorithm do not threaten our current invariants, and are therefore less complex to integrate.

Reaching a level of abstraction where the PTs are no longer present simplifies the whole model and alleviates the proof effort for the other dependent subsystems. Our abstract model can be extended to integrate additional features such as management of devices, that we have not taken into account in this paper, since the focus is on the SPT algorithm. More specifically, if we virtualize devices, the hypervisor controls the guest accesses to memory of the devices, so, in this case, we can ensure isolation. If we do not virtualize devices, every guest who has access rights to a device memory region can access it, in this case there might be channels between the guests accessing this part of memory. Another feature not currently accommodated by the hypervisor model is direct memory access (DMA) from devices. DMA hardware extensions (I/O MMU [2], SMMU [4,13]) allow the hypervisor to control the access to memory by a PT mechanism similar to the MMU, and can be proved secure. Without such extensions, DMA-aware devices can access any part of the memory and make it impossible to establish isolation within any model. Further work will investigate how to integrate device management and DMA into our models.

References

1. Alkassar, E., Cohen, E., Kovalev, M., Paul, W.J.: Verification of TLB virtualization implemented in C. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 209–224. Springer, Heidelberg (2012). http://www-wjp.cs.uni-saarland.de/publikationen/ACKP12.pdf
2. AMD I/O virtualization technology (IOMMU) specification (2015). http://support.amd.com/TechDocs/48882_IOMMU.pdf
3. Andrabi, S.J.: Verification of XMHF HPT protection setup. Technical report, University of North Carolina (2013). http://cs.unc.edu/~sandrabi/Project_work/VerificationofXMHFHPTProtectionSetup.pdf
4. ARM system memory management unit (2012). http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0062b/index.html

5. Arpaci-Dusseau, R.H., Arpaci-Dusseau, A.C.: Operating Systems: Three Easy Pieces, 0.80th edn. Arpaci-Dusseau Books, Wisconsin (2014)
6. Barthe, G., Betarte, G., Campo, J., Luna, C.: Cache-leakage resilient OS isolation in an idealized model of virtualization. In: 2012 IEEE 25th Computer Security Foundations Symposium (CSF), pp. 186–197, June 2012
7. Blanchard, A., Kosmatov, N., Lemerre, M., Loulergue, F.: A case study on formal verification of the anaxagoros hypervisor paging system with frama-C. In: Núñez, M., Güdemann, M. (eds.) Formal Methods for Industrial Critical Systems. LNCS, vol. 9128, pp. 15–30. Springer, Heidelberg (2015). http://dx.doi.org/10.1007/978-3-319-19458-5_2
8. Common criteria portal. http://www.commoncriteriaportal.org/
9. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: a practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009). http://research.microsoft.com/apps/pubs/default.aspx?id=117859
10. Daum, M., Billing, N., Klein, G.: Concerned with the unprivileged: user programs in kernel refinement. Formal Asp. Comput. **26**(6), 1205–1229 (2014). http://dx.doi.org/10.1007/s00165-014-0296-9
11. Kovalev, M.: TLB virtualization in the context of hypervisor verification. Ph.D. thesis, Universität des Saarlandes, Postfach 151141, 66041 Saarbrücken (2013). http://scidok.sulb.unisaarland.de/volltexte/2013/5215
12. Lescuyer, S.: ProvenCore: towards a verified isolation micro-kernel. In: International Workshop on MILS: Architecture and Assurance for Secure Systems (2015). http://milsworkshop2015.euromils.eu/downloads/hipeac_literature/04-mils15_submission_6.pdf
13. Mijat, R., Nightingale, A.: Virtualization is coming to a platform near you (2011). https://www.arm.com/files/pdf/System-MMU-Whitepaper-v8.0.pdf
14. Murray, T.C., Matichuk, D., Brassil, M., Gammie, P., Bourke, T., Seefried, S., Lewis, C., Gao, X., Klein, G.: sel4: from general purpose to a proof of information flow enforcement. In: 2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, 19–22 May 2013. pp. 415–429 (2013). http://dx.doi.org/10.1109/SP.2013.35
15. Nemati, H., Guanciale, R., Dam, M.: Trustworthy virtualization of the ARMv7 memory subsystem. In: Italiano, G.F., Margaria-Steffen, T., Pokorný, J., Quisquater, J.-J., Wattenhofer, R. (eds.) SOFSEM 2015-Testing. LNCS, vol. 8939, pp. 578–589. Springer, Heidelberg (2015). http://dx.doi.org/10.1007/978-3-662-46078-8_48
16. Vasudevan, A., Chaki, S., Jia, L., McCune, J., Newsome, J., Datta, A.: Design, implementation and verification of an extensible and modular hypervisor framework. In: Proceedings of the 2013 IEEE Symposium on Security and Privacy, SP 2013, pp. 430–444. IEEE Computer Society, Washington (2013). http://dx.doi.org/10.1109/SP.2013.36
17. Vetter, J., Petschik-Junker, M., Nordholz, J., Peter, M., Danisevskis, J.: Uncloaking rootkits on mobile devices with a hypervisor-based detector. In: ICISC (International Conference on Information Security and Cryptology), Seoul, Republic of Korea (2015)