Trustworthy Virtualization of the ARMv7 Memory Subsystem

Hamed Nemati, Roberto Guanciale, and Mads Dam

KTH Royal Institute of Technology, Stockholm, Sweden
{hnnemati,robertog,mfd}@kth.se

Abstract. In order to host a general purpose operating system, hypervisors need to virtualize the CPU memory subsystem. This entails dynamically changing MMU resources, in particular the page tables, to allow a hosted OS to reconfigure its own memory. In this paper we present the verification of the isolation properties of a hypervisor design that uses direct paging. This virtualization approach allows to host commodity OSs without requiring either shadow data structures or specialized hardware support. Our verification targets a system consisting of a commodity CPU for embedded devices (ARMv7), a hypervisor and an untrusted guest running Linux.The verification involves three steps: (i) Formalization of an ARMv7 CPU that includes the MMU, (ii) Formalization of a system behavior that includes the hypervisor and the untrusted guest (iii) Verification of the isolation properties. Formalization and proof are done in the HOL4 theorem prover, thus allowing to re-use the existing HOL4 ARMv7 model developed in Cambridge.

Keywords: formal verification, hypervisor, memory management.

1 Introduction

Memory isolation is a key requirement of systems executing software at different privilege levels. Inevitable security flaws of COTS OSes make providing trustworthy memory isolation using only OS level security mechanisms infeasible. Alternatively, memory isolation can be provided by leveraging special low-level execution platforms, like hypervisors. The small code-base of hypervisors makes the verification of their memory isolation properties feasible. In that way, a hypervisor can be used to host a commodity OS (which provides the non-critical services) along with several critical components, which are deployed in isolated partitions.

In this paper we present the formal verification of the memory isolation properties guaranteed by a hypervisor for embedded systems. Here, we focus on the main functionality of the hypervisor namely the virtualization of the memory subsystem, which determines the binding of physical memory locations to locations addressable at the application level. In order to properly isolate partitions, the hypervisor takes control of the memory configuration, by configuring the MMU and the corresponding page tables. Moreover, in order for such a hypervisor to host a general purpose OS it is necessary to allow the guest OS to

G.F. Italiano et al. (Eds.): SOFSEM 2015, LNCS 8939, pp. 578–589, 2015.
© Springer-Verlag Berlin Heidelberg 2015

dynamically reconfigure its internal memory hierarchy and to impose its own access restrictions. The virtualization of the memory subsystem must provide this functionality and play the role of a security monitor for the MMU settings. In fact, since the MMU is the key functionality used by the hypervisor to isolate the security domains, violation of complete mediation of the MMU settings can enable an attacker to bypass the hypervisor policies which could compromise the security of the entire system. This is also what makes a formal analysis of correctness a worthwhile enterprise.

A distinguishing feature of our work is the adoption of direct paging as virtualization mechanism. This mechanism has been previously introduced by Xen [6] and permits to virtualize the memory subsystem without requiring either shadow data structures (e.g., shadow page tables) or specialized hardware (e.g., nested page tables [2,6]). Direct paging allows a virtualization aware (i.e., paravirtualized) guest to directly manipulate the page tables while they are in passive state, i.e., not in active use by the MMU, and then using a dedicated API, verified in this paper, that effectuates and monitors the transition of page tables between passive and active state. Subsequent operations (like mapping specific entries) are done by invoking the corresponding hypercall, until the page tables are freed.

Our verification is done in the HOL4 theorem prover and targets a system consisting of a commodity CPU for embedded devices (ARMv7 [2]), which hosts the hypervisor and an untrusted guest. The verification is done in three steps: (i) We formalize (Section 2) the hosting hardware by extending the existing ARMv7 model developed in Cambridge with the formal model of the ARMv7 MMU, (ii) We formalize (Section 3) the behavior of the complete system by introducing a system model, which interleaves instructions executed by an untrusted guest with a low-level specification of the hypervisor handlers, and (iii) we prove (Sections 4 and 5) the security properties, by decomposing the proof into lemmas that can be reused by other virtualization mechanisms that use direct paging.

The verification is made complex by the level of abstraction. We target a real commodity CPU architecture and the model of the handlers is deliberately low level so that the implementation can be directly derived from the specification.

2 Formal Model of the ARMv7 CPU

In this model a machine state is modeled as a record $\sigma = \langle \mathit{regs}, \mathit{coregs}, \mathit{mem} \rangle \in \Sigma$, where regs, coregs and $\mathit{mem} \in 2^{32} \rightarrow 2^8$, respectively, represent the registers, coprocessors and system memory. In the state σ, the function $\mathit{mode}(\sigma)$ determines the current privilege execution mode, which can be either $PL0$ (user mode, used by the guest) or $PL1$ (privileged mode, used by the hypervisor). Here, the three coprocessor registers $\mathit{coregs} = \langle SCTLR, TTBR0, DACR \rangle \in 2^{32} \times 2^{32} \times 2^{32}$ are the System Control Register, the Translation Table Base Control Register, and the Domain Access Control Register respectively.

The system behavior is modeled by the state transition relation $\xrightarrow{l \in \{PL0, PL1\}} \subseteq \Sigma \times \Sigma$, where a transition realizes the effects of the execution of an ARM instruction. Non-privileged transitions ($\sigma \xrightarrow{PL0} \sigma'$) start and end in $PL0$ states.

$$\frac{\begin{array}{l}\sigma.SCTLR \neq 0\\ desc = read_{L_1}(\sigma.TTBR0, \sigma, va.l1_idx)\\ desc.type = SEC\\ (ap_check_{L1}(desc, PL, accreq))\end{array}}{mmu(\sigma, PL, va, accreq) = translate_{L1}(desc, va)}
\qquad
\frac{\begin{array}{l}\sigma.SCTLR \neq 0\\ desc_{L1} = read_{L_1}(\sigma.TTBR0, \sigma, va.l1_idx)\\ desc_{L1}.type = PT\\ desc_{L2} = read_{L_2}(desc_{L1}.pa, \sigma, va.l2_idx)\\ desc_{L2}.type = SP\\ (ap_check_{L2}(desc_{L2}, PL, accreq))\end{array}}{mmu(\sigma, PL, va, accreq) = translate_{L2}(desc_{L2}, va)}$$

Fig. 1. (a) One-step and (b) Two-step address translation

All the other transitions ($\sigma \xrightarrow{PL1} \sigma'$) involve at least one state in the privileged level. A transition from $PL0$ to $PL1$ is done by raising an exception, that can be caused by software interrupts, illegitimate memory accesses and hardware interrupts.

We extended the HOL4 ARMv7 model developed in Cambridge [3] to take into account the behavior of the ARMv7 MMU. The main functionalities of the MMU are virtual-to-physical address mapping and memory access control. The MMU is modeled by the $mmu(\sigma, PL, va, accreq) \to pa \cup \{\bot\}$ function. The function takes the state σ, a privilege level PL, a virtual address $va \in 2^{32}$ and the requested access right $accreq \in \{rd, wt, ex\}$, for *read*, *write* and *execution* respectively, and returns either the corresponding physical address $pa \in 2^{32}$ (if the access is granted) or an access permission fault ($\bot$).

The SCTLR register controls the MMU. If the MMU is disabled ($SCTLR = 0$), all access permissions are granted and the *mmu* function yields an identity mapping ($mmu(\sigma, PL, va, accreq) = va$). If the MMU is enabled ($SCTLR \neq 0$), the *mmu* can execute up to two page table walks to translate a virtual address. The first walk accesses the active first level ($L1$) page table, whose address is identified by the TTBR0 register.

An $L1$ page table contains 4096 entries, each mapping 1MB of contiguous virtual memory. When executing the first table walk, the MMU accesses the proper L1 entry. If the entry is *unmapped*, then the MMU yields a permission fault ($\bot$). If the entry is *Section* (SEC) (Figure 1a) and the requested access is legitimate ($ap_check_{L1}(desc, PL, accreq)$) then the corresponding 1MB is linearly mapped to 1MB of physical memory ($translate_{L1}(desc, va)$). If the entry is *Page table* (PT), then a second page table walk is needed (Figure 1b). In this case, the L1 entry points to an L2, which contains 256 entries. Each L2 entry, of type *Small page* (SP), linearly maps 4KB of virtual memory.

We introduce some auxiliary definitions to describe the properties guaranteed by an ARMv7 CPU that obeys the access privileges computed by the MMU. We use the predicate $mmu_{phys}(\sigma, PL, pa, accreq)$ to identify the accesses granted to the physical memory. An access to a physical address is allowed if at least one virtual address exists that enables the requested access permission and that maps to pa; $mmu_{phys}(\sigma, PL, pa, accreq) = (\exists va.\ mmu(\sigma, PL, va, accreq) = pa)$.

We say that two states are *MMU-consistent* if their memories differ only for writable physical addresses. Formally, $mmu_c(\sigma, \sigma', PL)$ if $\forall pa.\ \ \sigma.mem(pa) \neq \sigma'.mem(pa) \Rightarrow mmu_{phys}(\sigma, PL, pa, wt)$.

Two states are *MMU-equivalent* if for any virtual address va the MMU yields the same translation and the same access permissions. Formally, $\sigma \overset{mmu}{\equiv} \sigma'$ if and only if $\forall va, PL, accreq.\ \ mmu(\sigma, PL, va, accreq) = mmu(\sigma', PL, va, accreq)$.

The transition relation queries the MMU model to identify when an instruction raises an exception and satisfies the following properties.

Property 1. Let $\sigma \in \Sigma$ such that $mode(\sigma) = PL0$. If $\sigma \xrightarrow{PL0} \sigma'$ then $mmu_c(\sigma, \sigma', PL0)$ and $\sigma.coregs = \sigma'.coregs$.

Property 2. Let $\sigma_1, \sigma_2 \in \Sigma$, A be a set of physical addresses, $mode(\sigma_1) = mode(\sigma_2) = PL0$ and $A \supseteq \{pa \mid \exists accreq.\ mmu_{phys}(\sigma, PL0, pa, accreq)\}$, and $\sigma_1 \overset{mmu}{\equiv} \sigma_2$, $\sigma_1.regs = \sigma_2.regs$, $\sigma_1.coregs = \sigma_2.coregs$ and $\forall pa \in A.\ \sigma_1.mem(pa) = \sigma_2.mem(pa)$. If $\sigma_1 \xrightarrow{PL0} \sigma_1'$ and $\sigma_2 \xrightarrow{PL0} \sigma_2'$ then $\sigma_1'.regs = \sigma_2'.regs$, $\sigma_1'.coregs = \sigma_2'.coregs$ and $\forall pa \in A.\ \sigma_1'.mem(pa) = \sigma_2'.mem(pa)$.

In [8] the authors validated the HOL4 ARMv7 model against these properties under the assumption that the address translation is the identity map.

3 Formal Model of the Hypervisor

Our target scenario consists of an ARMv7 CPU hosting the hypervisor and an untrusted guest. The hypervisor uses direct paging to virtualize the memory subsystem: the page tables are allocated by the guest inside its own memory and the guest is allowed to manage its page table while tables are not in active use by the MMU. Subsequent operations on the tables are done by invoking the corresponding hypercall serving the request. In our approach the physical memory is logically fragmented into blocks of 4KB, resulting in 2^{20} possible physical blocks. Since L1 and L2 page tables are 16KB and 1KB respectively, an L1 page table is stored in four contiguous physical blocks and a physical block can contain four L2 page tables. The hypervisor associates a type to each block: (i) D; the block does not contain sensitive data, (ii) L_1: the block contains part of an L1 page table, and (iii) L_2: the block contains four L2 page tables.

To handle guest requests, the hypervisor provides nine hypercalls: *switch* that selects the active L1, *L1create* and *L2create* to change the type of a block to the corresponding table type, *L1free* and *L2free* to free the page tables and changing the type of a block to D, *L1unmap* and *L2unmap* to unmap a page table entry, *L1map* and *L2map* to map a specific entry of a page table.

The hypercalls enforce the page type policy: the guest is allowed to change only blocks of type D. Naively enforcing this policy requires the hypervisor to re-validate the page tables before reactivating them, that is a time consuming task. To make overhead sustainable, the hypervisor maintains a reference counter for each block. The intuition is that the hypervisor changes the type of a physical block (e.g., allocates or frees a page table) only if the corresponding reference counter is zero and that this enables the hypervisor to skip the re-validation tasks.

We model the complete system reusing the formal model of Section 2. A system state is modeled by a tuple $\langle \sigma, h \rangle$, consisting of an ARMv7 state σ

$$\frac{\begin{array}{l} bls = \{block(pa) + i \mid i < 4\} \\ \forall bl \in bls. \tau \vdash bl : D \wedge G_{mem} \vdash bl : 0 \wedge \rho(bl) = 0 \\ descs = [read_{L_1}(pa, \sigma, j) \mid j < 4096] \\ pts = [block(d.pa) \mid d \in descs \wedge d.type = PT] \\ secs_{rd} = [block(d.pa) \mid d \in descs \wedge d.type = SEC \wedge (0, rd) \in d.ap] \\ secs_{wt} = [block(d.pa) \mid d \in descs \wedge d.type = SEC \wedge (0, wt) \in d.ap] \\ \forall bl \in pts. \tau \vdash bl : L_2 \\ \forall bl \in secs_{rd}. \forall idx < 256. G_{mem} \vdash bl + idx : 0 \\ \forall bl \in secs_{wt}. \forall i < 2^8. bl + i \notin bls \wedge \tau \vdash bl + i : D \wedge \rho(bl + i) < MAX - 2^{12} \\ \rho' = for\ bl \in secs_{wt}(for\ i < 2^8\ (\lambda \rho_1. \rho_1(bl + i) := \rho_1(bl + i) + 1))\rho \\ \rho'' = for\ bl \in pts(\lambda \rho_2. \rho_2(bl) := \rho_2(bl) + 1)\rho' \\ \tau' = (\tau(bl \in bls) := L_1) \end{array}}{\langle \sigma, \langle \tau, \rho \rangle \rangle \xrightarrow{createL1(pa)} \langle \sigma, \langle \tau', \rho'' \rangle \rangle}$$

Fig. 2. Inference rule for hypervisor *createL1* handler

and an abstract hypervisor state h, of the form $\langle \tau, \rho \rangle$. Let $bl \in 2^{20}$ be the index of a physical block and $t \in \{D, L1, L2\}$, $\tau \vdash bl : t$ tracks the type of the block and $\rho(\tau) \in 2^{30}$ tracks the reference counter. We use G_{mem} to statically identify the memory region assigned to the guest: if the block bl is part of the guest memory then $G_{mem} \vdash bl : 0$, otherwise $G_{mem} \vdash bl : 1$

The behavior of the system is defined by a labeled transition relation $\langle \sigma, h \rangle \xrightarrow{\alpha} \langle \sigma', h' \rangle$. The model interleaves standard non-privileged transitions ($\alpha = 0$) with abstract handler invocations (e.g., $\alpha = createL1(pa)$):

- if $\sigma \xrightarrow{PL0} \sigma'$ then $\langle \sigma, h \rangle \xrightarrow{0} \langle \sigma', h \rangle$; instructions executed in non-privileged mode that do not raise exceptions behave equivalently to the standard ARMv7 semantics and do not affect the abstract hypervisor state.
- if $\sigma \xrightarrow{PL1} \sigma'$ then the hypervisor intercepts the exception and its handler atomically transforms the state of the system

The model of the hypervisor handlers is defined by HOL4 functions that describe the hypervisor behavior. These functions are deliberately low level, for example they store the page tables in the system memory instead of using abstract data structures. This enables us to include the page tables in the attack surface taken into account by our verification. The hypervisor handlers check that (i) guest can only change *data* pages (pages of type D), (ii) page table blocks are typed correctly, and (iii) the blocks that are readable/writable by the guest are enclosed in the part of the memory granted to the guest. If a handler fails to validate the guest request, then it terminates without affecting the system state (i.e., $\langle \sigma, \langle \tau, \rho \rangle \rangle \xrightarrow{\alpha} \langle \sigma, \langle \tau, \rho \rangle \rangle$.

In Figure 2 we use an inference-rule to exemplify the behavior of the *L1create* handler. The guest uses the hypercall to request the validation of an L1 pointed to by the address pa. If the validation succeeds the type of the corresponding physical blocks is changed to $L1$. Here, $block(pa)$ returns the block pointed to by the physical address pa and $[f(x) \mid p(x)]$ represents list comprehension. Moreover, $r(\beta) := value$ represents the update of the field β of the record r with $value$. Finally, let d be an entry of a page table, we use $d.pa$, $d.type$ and $d.ap$ to represent the initial physical address pointed to by the entry, its type (either a section SEC or a page table PT) and the set of the granted access rights.

The handler checks that the four blocks containing the new page table have reference zero, are typed D and reside in the guest memory. To accept the request, each PT entry must point to a valid $L2$ in the guest memory. Section descriptors allowing guest read accesses must point to a section encompassing only blocks that are part of the guest memory. Moreover, if the descriptor allows guest write access, then the hypervisor ensures that all the reachable blocks are typed D, that their reference counter is less than the maximum allowed bound and that none of the blocks of the new page table are included in that section.

The hypercall also updates the reference counters of the pointed blocks, by summing the number of references that enable guest write access and the number of references from PT entries.

4 Security Properties

As common, our verification strategy consists in introducing a state invariant ($\mathcal{I}(s)$) that guarantees the desired security properties and in demonstrating that the invariant is preserved by any possible transition. Clearly, the system must start (i.e., the boot must terminate) in a state that satisfies the invariant. We use $\mathcal{Q}_\mathcal{I}$ to identify the set of all possible states that satisfy the invariant.

Theorem 1. *Let $s \in \mathcal{Q}_\mathcal{I}$, if $s \xrightarrow{\alpha} s'$ then $s' \in \mathcal{Q}_\mathcal{I}$.*

Guaranteeing that the invariant is preserved requires to demonstrate that each handler preserves the invariant ($\alpha \neq 0$) and that the guest is not able to break it. Intuitively, while the hypervisor is inactive ($\alpha = 0$), the only mechanism that can confine the behavior of an arbitrary guest is the MMU. Thus, the hypervisor must play the role of a security monitor of the MMU settings. If *complete mediation* of the MMU settings is violated, then an attacker can bypass the hypervisor policies and compromise the security of the entire system. Enforcing this property is critical in the direct paging mechanism because the page tables are dynamically allocated and released by the untrusted guest and they reside in the guest memory.

Theorem 2. *Let $s \in \mathcal{Q}_\mathcal{I}$, if $s \xrightarrow{0} s'$ then $s \overset{mmu}{\equiv} s'$.*

Definition 1. *Two states s and s' do not differ in a physical block of memory (written $s \overset{bl}{\equiv} s'$) if for each physical address pa if $block(pa) = bl$ then $s.mem(pa) = s'.mem(pa)$.*

Definition 2. *Two states s and s' are t-equivalent (written $s \overset{G_{mem}:t}{\equiv} s'$) iff for each physical block bl if $G_{mem} \vdash bl : t$ then $s \overset{bl}{\equiv} s'$.*

We use the approach of [5] to analyze the data separation properties. The *non-exfiltration* property guarantees that a transition executed by the guest does not modify the secure resources:

Theorem 3. *Let* $s \in \mathcal{Q}_{\mathcal{I}}$*, if* $s \xrightarrow{0} s'$ *then* $s \overset{G_{mem}:1}{\equiv} s'$*.*

The *non-infiltration* property is a noninterference property guaranteeing that a transition executed by the guest depends only on its accessible resources.

Theorem 4. *Let* $s_1, s_2 \in \mathcal{Q}_{\mathcal{I}}$ *such that* $s_1.regs = s_2.regs$*,* $s_1.coregs = s_2.coregs$ *and* $s_1 \overset{G_{mem}:0}{\equiv} s_2$*. If* $s_1 \xrightarrow{\alpha} s_1'$ *and* $s_2 \xrightarrow{\alpha} s_2'$ *then* $s_1'.regs = s_2'.regs$*,* $s_1'.coregs = s_2'.coregs$ *and* $s_1' \overset{G_{mem}:0}{\equiv} s_2'$

5 Verification Strategy

To describe the verification of the security properties we summarize the structure of the system invariant.

Definition 3. $\mathcal{I}(\langle \sigma, \langle \tau, \rho \rangle \rangle)$ *holds if,*

$$\sigma.SCTLR \neq 0 \wedge \tau \vdash \sigma.TTBR0 : L_1 \wedge \forall bl \in 2^{20}.\ \mathcal{I}_T(\langle \sigma, \langle \tau, \rho \rangle \rangle, bl) \wedge \mathcal{I}_C(\langle \sigma, \langle \tau, \rho \rangle \rangle, bl)$$

$$\text{where } \mathcal{I}_C(\langle \sigma, \langle \tau, \rho \rangle \rangle, bl) \text{ holds if, } \rho(bl) = \sum_{i \in 0 \ldots 2^{20}} cnt(\sigma, \tau, bl, i)$$

$$\text{and } \mathcal{I}_T(\langle \sigma, \langle \tau, \rho \rangle \rangle, bl) \text{ holds if, } \begin{cases} \tau \vdash bl : L_1 \Rightarrow \mathcal{I}_{T_1}(\sigma, \tau, bl) \\ \tau \vdash bl : L_2 \Rightarrow \mathcal{I}_{T_2}(\sigma, \tau, bl) \end{cases}$$

The invariant requires that the MMU is enabled, the active page table is typed L_1, the reference counter is correctly counting the references to each block and each block is well-typed.

We use Figure 3 to summarize the properties checked by the invariant. The table in the center represents the physical memory and reports the page type (pt), the static type (gm) and the reference counter of each block (rc).

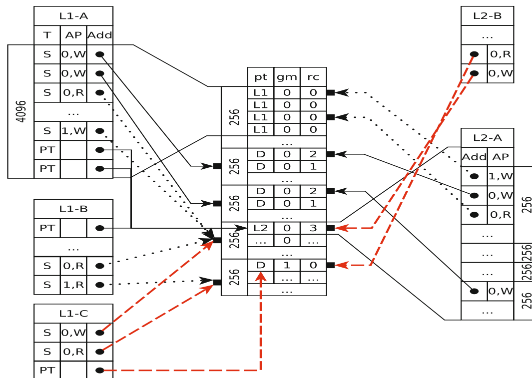

Fig. 3. Invariant

The four top most blocks contain an L1, whose 4096 entries are depicted by the table L1-A. The top entry is a section descriptor ($T = S$) that grants write permission to the guest ($ap = 0, w$). The entry points to the second physical section, which consists of 256 blocks. Three other section descriptors are depicted: one grants write accesses to the guest, one grants only read permission to the guest $(0, r)$, and the last one prevents any guest access and enables write permission to the privileged mode $(1, w)$. The last two entries of the L1 are PT-entries. These two entries point to two different L2 page tables that are stored in the same physical block. To satisfy $\mathcal{I}_{T_1}(\sigma, \tau, 0)$, if a section enables guest access then the pointed blocks must all be in the guest memory. Moreover, if a section enables guest write access then each pointed block must be typed D. Finally, PT-entries must point to addresses contained in physical blocks that are typed L_2. The Figure depicts two additional L1 page tables; L1-B satisfies the invariant, L1-C contains three entries that violate $\mathcal{I}_{T_1}$. In fact (i) the first section grants write permission to the guest, but at least one of the pointed blocks is not typed D, (ii) the second section enables guest accesses, but at least one of the pointed blocks is not in the guest memory and (iii) the third entry is a PT-entry, but points to a physical address that is contained by a block typed D.

The table L2-A depicts the content of the 768-th physical block, which contains four L2 page tables. To satisfy $\mathcal{I}_{T_2}(\sigma, \tau, 768)$, if one entry of the four page tables enables guest access then the pointed block must be in the guest memory. Moreover, if the entry also enables guest write access then the pointed block must be typed D.

We use the same figure to illustrate the reference counter. For a physical block bl, if block i is typed L_1 then $cnt(\sigma, \tau, bl, i)$ counts the number of section entries that point to bl and that are writable in user mode plus the number of PT-entries that point to bl. If block i is typed L_2 then $cnt(\sigma, \tau, bl, i)$ counts the number of entries that point to bl and that are writable in user mode. In Figure 3 we use solid arrows to represent the references that are counted and dashed arrows to represent the other references.

Lemma 1. *Let* $\langle \sigma, \langle \tau, \rho \rangle \rangle \in \mathcal{Q}_\mathcal{I}$ *then* $\forall pa.\ mmu_{ph}(\sigma, 0, pa, wt) \Rightarrow (G_{mem} \vdash block(pa) : 0 \wedge \tau \vdash block(pa) : D)$

Lemma 2. *Let* $\langle \sigma, \langle \tau, \rho \rangle \rangle \in \mathcal{Q}_\mathcal{I}$ *then* $\forall pa.\ mmu_{ph}(\sigma, 0, pa, rd) \Rightarrow (G_{mem} \vdash block(pa) : 0)$

Lemmas 1 and 2 directly follow from the invariant and show that the MMU setup forbids guest accesses outside the guest memory and guest write accesses to physical blocks that are not typed D. Using Lemma 1 and Property 1 we directly show that the guest can only change physical blocks that are inside its own memory and that are typed D.

Lemma 3. *Let* $s = \langle \sigma, \langle \tau, \rho \rangle \rangle \in \mathcal{Q}_\mathcal{I}$ *and* $s \xrightarrow{0} s'$. *For each* bl, *if* $s \overset{bl}{\not\equiv} s'$ *then* $G_{mem} \vdash bl : 0$ *and* $\tau \vdash bl : D$.

Notice that Lemma 3 directly guarantees Theorem 3. Similarly, Lemma 1 and Property 2 guarantee Theorem 4 for guest transitions (i.e., $\alpha = 0$).

Proof of Theorem 2. We must ensure that for each possible address va the MMU translation is equivalent in the states $s = \langle \sigma, \langle \tau, \rho \rangle \rangle$ and $s' = \langle \sigma', \langle \tau, \rho \rangle \rangle$. Property 1 guarantees that the coprocessors registers are not affected by user transitions, thus the address of the active L1 can not be changed by the guest. The first translation walk accesses the L1 entry that maps va. The invariant $\mathcal{I}(s)$ guarantees that the active page L1 is typed L_1 ($\tau \vdash \sigma.c2 : L_1$). Thus, Lemma 3 guarantees that in the two states s and s' the MMU fetches the same L1 descriptor. If the descriptor is either unmapped or a section the proof is completed. If the descriptor is a PT-entry then the translation executes the second walk, accessing an entry of the pointed $L2$. Let bl be the block containing the pointed $L2$. From the invariant, we know that $\tau \vdash bl : L_2$. Again, Lemma 3 guarantees that in the two states s and s' the MMU fetches the same L2 descriptor, thus concluding the proof.

Lemma 4. *Let* $\langle \sigma, \langle \tau, \rho \rangle \rangle \in \mathcal{Q}_{\mathcal{I}}$, *if* $\sigma \overset{bl}{\equiv} \sigma'$ *then* $\mathcal{I}_T(\langle \sigma', \langle \tau, \rho \rangle \rangle, bl)$ *and* $\forall bl'.\ cnt(\sigma, \tau, bl', bl) = cnt(\sigma', \tau, bl', bl)$.

Lemma 4 shows that the well-typeness of a block and its counted references are independent from the content of the other physical blocks.

Lemma 5. *Let* $s \in \mathcal{Q}_{\mathcal{I}}$, *if* $s \xrightarrow{0} s'$ *then* $s' \in \mathcal{Q}_{\mathcal{I}}$.

Lemma 5 shows that the invariant is preserved by the execution of an arbitrary guest instruction, thus guaranteeing that untrusted software can not breach the security of the system. Since the guest can not directly change the abstract hypervisor data-structures and can not directly affect the coprocessor registers (Property 1), in order to reestablish the invariant we need to show that for every block bl both $\mathcal{I}_T(\langle \sigma', \langle \tau, \rho \rangle \rangle, bl)$ and $\mathcal{I}_C(\langle \sigma', \langle \tau, \rho \rangle \rangle, bl)$ hold. If $\tau \vdash bl : D$ then $\mathcal{I}_T$ trivially holds. Otherwise, Lemma 3 guarantees that $s \overset{bl}{\equiv} s'$ and Lemma 4 demonstrates that $\mathcal{I}_T$ holds. For the reference counter we must prove that $\rho(bl) = \sum_{i \in 0 \ldots 2^{20}} cnt(\sigma', \tau, bl, i)$ knowing that $\rho(bl) = \sum_{i \in 0 \ldots 2^{20}} cnt(\sigma, \tau, bl, i)$. We directly show that for every block i $cnt(\sigma', \tau, bl, i) = cnt(\sigma, \tau, bl, i)$. If $\tau \vdash i : D$ this equality is trivial, since cnt is zero. Otherwise, Lemma 3 guarantees that $s \overset{i}{\equiv} s'$ and Lemma 4 concludes the proof.

Lemma 6. *Let* $\langle \sigma, \langle \tau, \rho \rangle \rangle \in \mathcal{Q}_{\mathcal{I}}$ *and* $\forall bl'.\ (\tau(bl') \neq \tau'(bl')) \Rightarrow (\rho(bl') = 0)$. *For every* bl *if* $\tau'(bl) = \tau(bl)$ *then* $\mathcal{I}_T(\langle \sigma, \langle \tau', \rho' \rangle \rangle, bl)$ *and* $\forall bl'.\ cnt(\sigma, \tau, bl', bl) = cnt(\sigma, \tau', bl', bl)$.

Lemma 6 expresses that well-typedness and counters are preserved for all hypervisor data changes, as long as the blocks whose types change have reference counter zero. The equality of the reference counter is established by showing that cnt is independent on the type of the block bl'. The strategy used to establish $\mathcal{I}_T$ depends on the type of the block bl. If $\tau \vdash bl : D$ then the proof is trivial. If $\tau \vdash bl : L_2$ we use reductio ad absurdum. Assume that $\mathcal{I}_T$ does not hold, then there must exist an entry i of the page table that grants guest write access to

a block bl' that is not typed D in τ'. From the invariant, we know that such an entry does not exist in $\langle \sigma, h \rangle$. Thus, i must point to a block bl' such that $(\tau(bl') \neq \tau'(bl'))$. The hypothesis of the Theorem guarantees that $\rho(bl') = 0$, then for all bl'' (including bl) $cnt(\sigma, \tau, bl', bl'') = 0$. This contradicts the assumption that there is an entry i that points to bl' and that is writable by the guest. For $\tau \vdash bl : L_1$ we use a similar reasoning.

Lemmas 4 and 6 are used to modularize the proof of Theorem 1 for the transitions modeling the hypervisor handlers. For example, to demonstrate that the invariant is preserved by $createL1(pa)$ we first show that the type is changed only for pages having reference zero, then we demonstrate that the content of the blocks containing the page tables is not changed.

For the $\mathcal{I}_T$ predicate, we use the two Lemmas to guarantee that $\mathcal{I}_T$ is preserved for blocks that are not in $bls = \{block(pa) + i \mid i < 4\}$. Then, we demonstrate that the checks performed by the hypercall guarantees $\mathcal{I}_{T_1}(\sigma', \tau', bl)$ for the every block in bls (i.e., the blocks containing the new L1).

For the $\mathcal{I}_C$ predicate, since the invariant guarantees that $\rho(bl) = \sum_{i \in 0 \ldots 2^{20}} cnt(\sigma, \tau, bl, i)$, then we must prove :

$$\rho'(bl) = \textstyle\sum_{i \in 0 \ldots 2^{20}} cnt(\sigma', \tau', bl, i) + \rho(bl) - \sum_{i \in 0 \ldots 2^{20}} cnt(\sigma, \tau, bl, i).$$

For a block i that is not in bls, the two Lemmas guarantee that $cnt(\sigma', \tau', bl, i) = cnt(\sigma, \tau, bl, i)$, thus our goal is reduced to demonstrate $\rho'(bl) = \sum_{i \in bls} cnt(\sigma', \tau', bl, i) + \rho(bl) - \sum_{i \in bls} cnt(\sigma, \tau, bl, i)$. Since the hypercall checks that the initial type of a block $i \in bls$ is D, then $cnt(\sigma, \tau, bl, i) = 0$. Finally, we must demonstrate that the hypercall correctly updates the counters by adding the references of the new L1 page table: $\rho'(bl) = \rho(bl) + \sum_{i \in bls} cnt(\sigma', \tau', bl, i)$.

6 Evaluation

The verification has been performed using the HOL4 interactive theorem prover. This allows building the formal model of the system on top of the existing ARMv7 model developed in Cambridge [3], by extending the transition relation to take into account the MMU constraints and by substituting the activation of exceptions with the specification of the hypervisor handlers. This specification consists of 500 lines of HOL4 code, intentionally avoids any high level construct and resembles the control flow of the C implementation, with the aim of making this executable specification as close as possible to the C implementation. This increased the difficulty of the proof (e.g., the system invariant consists of 2k lines of HOL4 and the proofs consist of 15k lines of HOL4), which must handle finite arithmetic overflows, page tables not stored into abstract states and forced us to identify the invariants of the loops required to iterate the page tables. However, the low level of abstraction allowed us to identify several bugs of the original design: (i) arithmetic overflow when updating the reference counter, due to the guest being able to create unlimited references to a physical block (ii) bit field and offset mismatch, (iii) missing check that a newly allocated page table prevents the guest from overwriting the page table itself, (iv) usage of the

signed shift operator where the unsigned one was necessary, and (v) side channels exploitable by the guest by requesting to validate page tables outside the guest memory.

We implemented a prototype of the hypervisor to experiment the verified design. This prototype consists of 4500 LOC of C (with a minor part of assembly) and is directly derived from the verified specification. Initial benchmarks show promising results; the hypervisor is capable of hosting a paravirtualized Linux and for the LMBench introduces an overhead between 2% (select benchmark) and 495% (fork+bin benchmark), compared to the native Linux. This overhead is much less than what the many other hypervisors for ARM impose [7].

We briefly compare our virtualization mechanism to existing approaches to virtualize the memory subsystem. Functional correctness of mechanisms based on Shadow page tables (SPT) has been verified in [1,4,10]. Using SPT, the hypervisor keeps a shadow copy of the guest page tables. This copy is updated (after validation) by the hypervisor whenever the guest operates on its page tables. The main benefits of direct paging respect to SPT is that the hypervisor does not replicate the guest page tables, thus reducing memory accesses and memory overhead and not requiring dynamic allocation in the hypervisor.

Hardware-assisted virtualization (e.g., nested-paging included in the ARMv7 virtualization extension) frees the hypervisors from implementing a virtualization mechanism of the memory subsystem (e.g., [6,11]). This simplification comes at the cost of enlarging the TCB and moving the verification from software to hardware. Moreover, since hardware virtualization support is still uncommon in embedded systems, then software based virtualization is the only viable option for several platforms (including ARM11, ARM CortexA5 and Intel Quark).

The formal verification of seL4 [9] demonstrated that the verification of a complete microkernel is possible even at the machine code level [12]. A complete commodity OS can be executed on top of a microkernel by mapping the OS threads directly to the microkernel threads, thus delegating completely the process management functionality from the hosted OSes to the microkernel (e.g., L^4Linux). This generally involves an invasive and error-prone OS adaptation process, however. An alternative approach consists of extending the microkernel with a virtualization mechanism of the memory subsystem, like the one proposed in this paper.

7 Concluding Remarks

We presented a design to virtualize the ARMv7 memory subsystem that requires neither specialized hardware support nor shadow data structures. Together with the machine-assisted proof of its correctness and the spatial isolation provided by the hypervisor, the design represents the first trustworthy virtualization mechanism using *direct paging*, previously introduced by Xen for x86, but not verified.

The design correctness is stated in terms of (i) complete mediation of the MMU settings, (ii) non-exfiltration and (iii) non-infiltration. These properties show that user mode processes (e.g., a possibly malicious guest OS) are incapable

of affecting the MMU behaviour and that can not influence (or be influenced by) the resources that are not allocated to the guest.

The low level abstraction of the hypervisor specification increased the complexity of our verification, but allowed us to identify and to correct several bugs. Moreover, since the specification avoids any high level constructs, it has been used to directly drive a prototype.

References

1. Alkassar, E., Hillebrand, M.A., Paul, W.J., Petrova, E.: Automated verification of a small hypervisor. In: Leavens, G.T., O'Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 40–54. Springer, Heidelberg (2010)
2. ARMv7-A architecture reference manual, `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406b`
3. Fox, A., Myreen, M.O.: A trustworthy monadic formalization of the aRMv7 instruction set architecture. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 243–258. Springer, Heidelberg (2010)
4. Heiser, G., Leslie, B.: The OKL4 microvisor: convergence point of microkernels and hypervisors. In: Thekkath, C.A., Kotla, R. and Zhou, L. (eds.), ApSys, pp. 19–24. ACM (2010)
5. Heitmeyer, C., Archer, M., Leonard, E., McLean, J.: Applying formal methods to a certifiably secure software system. IEEE Trans. Softw. Eng. 34(1), 82–98 (2008)
6. Hwang, J.-Y., Suh, S.-B., Heo, S.-K., Park, C.-J., Ryu, J.-M., Park, S.-Y., Kim, C.-R.: Xen on ARM: System virtualization using Xen hypervisor for ARM-based secure mobile phones. In: 5th IEEE Consumer Communications and Networking Conference, CCNC 2008, pp. 257–261. IEEE (2008)
7. Iqbal, A., Sadeque, N., Mutia, R.I.: An overview of microkernel, hypervisor and microvisor virtualization approaches for embedded systems. Report, Department of Electrical and Information Technology, Lund University, Sweden, 2110 (2009)
8. Khakpour, N., Schwarz, O., Dam, M.: Machine assisted proof of aRMv7 instruction level isolation properties. In: Gonthier, G., Norrish, M. (eds.) CPP 2013. LNCS, vol. 8307, pp. 276–291. Springer, Heidelberg (2013)
9. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an OS kernel. In: Matthews, J.N., Anderson, T.E. (eds.), SOSP, pp. 207–220. ACM (2009)
10. Leinenbach, D., Santen, T.: Verifying the Microsoft Hyper-V hypervisor with VCC. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 806–809. Springer, Heidelberg (2009)
11. McCoyd, M., Krug, R.B., Goel, D., Dahlin, M., Young, W.D.: Building a hypervisor on a formally verifiable protection layer. In: HICSS, pp. 5069–5078. IEEE (2013)
12. Sewell, T.A.L., Myreen, M.O., Klein, G.: Translation validation for a verified os kernel. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 471–482. ACM (2013)