Trustworthy Memory Isolation of Linux on Embedded Devices

Hamed Nemati, Mads Dam, Roberto Guanciale, Viktor Do, and Arash Vahidi

KTH Royal Institute of Technology, Stockholm, Sweden
{hnnemati,mfd,robertog}@kth.se
SICS Swedish ICT, Lund, Sweden
{viktordo,arash}@sics.se

Abstract. The isolation of security critical components from an untrusted OS allows to both protect applications and to harden the OS itself, for instance by run-time monitoring. Virtualization of the memory subsystem is a key component to provide such isolation. We present the design, implementation and verification of a virtualization platform for the ARMv7-A processor family. Our design is based on direct paging, an MMU virtualization mechanism previously introduced by Xen for the x86 architecture, and used later with minor variants by the Secure Virtual Architecture, SVA. We show that the direct paging mechanism can be implemented using a compact design, suitable for formal verification down to a low level of abstraction, without penalizing system performance. The verification is performed using the HOL4 theorem prover and uses a detailed model of the ARMv7-A ISA, including the MMU. We prove memory isolation of the hosted components along with information flow security for an abstract top level model of the virtualization mechanism. The abstract model is refined down to a HOL4 transition system closely resembling a C implementation. The virtualization mechanism is demonstrated on real hardware via a hypervisor capable of hosting Linux as an untrusted guest. [1]

1 Introduction

A basic security requirement for systems that allow software to execute at different levels of security is memory isolation: The ability to store secret information within a designated part of memory and prevent the contents of this memory to be affected by, or leaked to, parts of the system that are not authorized to access it. Without the usage of special hardware, trustworthy memory isolation relies on the correct implementation of the OS kernel. However, given the size and complexity of modern OSs, the vision of comprehensive and formal commodity OS verification is as distant as ever.

An alternative to verifying the entire OS is to delegate critical functionality to special low-level execution platforms such as hypervisors, separation kernels,

[1] Acknowledgement.Work supported by framework grant “IT 2010” from the Swedish Foundation for Strategic Research.

or microkernels. Such an approach has some significant advantages. First, the size and complexity of the execution platform can be made much smaller, potentially opening up for rigorous verification. The literature has many recent examples of this, in seL4 [16], Microsoft's Hyper-V project [17], Green Hills' CC certified INTEGRITY-178B separation kernel [22], and the PROSPER separation kernel [10]. Second, the platform can be opened up to public scrutiny and certification, independent of application stacks. Virtualization-like mechanisms can also be used to support various forms of application hardening against untrusted OSs. Examples of this include KCoFi [7] based on the Secure Virtual Architecture [9], Overshadow [5], Inktag [14], and Virtual Ghost [8]. All these cases rely crucially on memory isolation to provide the required security guarantees, typically by virtualizing the memory management unit (MMU) hardware. MMU virtualization, however, can be exceedingly tricky to get right, motivating the use of formal methods for its verification.

In this paper we present an MMU virtualization API for the ARMv7 family of processors (which is one of the widely adopted architectures for embedded devices) that has been formally verified down to a low level of abstraction. The API uses direct paging, a virtualization mechanism introduced by Xen [4] and used later with some variations by the Secure Virtual Architecture [9]. In direct paging, page tables are kept in guest memory and allowed to be read and directly manipulated by the untrusted guest OS (when they are not in active use by the MMU). Xen demonstrated that this approach has better performance than other software virtualization approaches (e.g. shadow page tables) on the x86 architecture [4]. Moreover, since direct paging does not require shadow data structures, this approach has small memory overhead. The engineering challenge we posed ourselves was to design a minimal API that is (i) sufficiently expressive to host a paravirtualized Linux, (ii) introduces an acceptable overhead and (iii) whose implementation is sufficiently small to be subject to pervasive verification for commodity CPU architecture such as ARMv7.

The security objective is to allow a malicious guest system to operate freely, invoking the hypervisor at will, without being able to access memory or processor resources that the guest has not received static permission for. The verification is performed using a formal model of the ARMv7 architecture [11], implemented in the HOL4 interactive theorem prover.

The verification is built on top a model, the top level specification (TLS), which describes the ideal behavior of hypervisor's handlers implementing the virtualization mechanism, alternating with user mode execution under control of a possibly malicious guest. Parts of the security state is stored in a model state, by construction outside the reach of the guest. However, page tables are stored in memory. This is a key complication forced by the direct paging approach, and the solution to this problem is a key contribution of the paper. The upshot is that it is no longer self-evident that the desired memory isolation properties, non-exfiltration and non-infiltration in the terminology of [13], hold for the TLS, and an important part of the verification is therefore to formally validate this fact.

To keep the TLS as simple and abstract as possible, the TLS addresses page tables directly using their physical addresses. A real implementation cannot do this, but must appeal to virtual addresses instead, in addition to managing its internal data structures. To this end we introduce an implementation model, essentially a state transition model operating on the real ARMv7-A state through transitions that directly reflect handler execution at the binary level. We exhibit a refinement from the TLS to the implementation model, prove its correctness, and show, as a corollary, that the memory isolation properties proved at top level transfer to the implementation level.

The verification highlighted three classes of bugs in the initial design of the virtualization mechanism:

(i) Arithmetic overflows, bit field and offset mismatches, and signed operators where the unsigned ones were needed.
(ii) Missing checks of self referencing page tables.
(iii) Approval of guest requests that cause unpredictable behaviors of the ARMv7 MMU.

Moreover, the verification of the implementation model identified additional bugs exploitable by requesting the validation of physical blocks residing outside the guest memory. This last class of bugs was identified because the implementation model takes into account the virtual memory mapping used by the handlers.

We report on a port of Linux kernel 2.6.34 and demonstrate the prototype implementation of a hypervisor for which the core component is the verified MMU virtualization API. Experiments demonstrate that the hypervisor can run with reasonable performance on real hardware (Beagleboard-xM based on the Cortex-A8 CPU).

2 Related Work

The ability to isolate security critical components from an untrusted OS allows non critical parts of a system to be implemented while the critical software remains adequately protected. This isolation can be used both to protect applications from an untrusted OS as well as to protect the OS itself from internal threats. For example, KCoFI [7] uses Secure Virtual Architecture [9] to isolate the OS from a run-time checker. The checker instruments the OS and monitors its activities to guarantee the control-flow integrity of the OS itself. Related examples are application hardening frameworks such as Overshadow [5], Inktag [14], and virtual ghost [8]. In all these cases some form of virtualization of the MMU hardware is a critical component to provide the required isolation guarantees.

Shadow page tables (SPT) is a common approach to MMU virtualization. The virtualization layer maintains a shadow copy of page tables created and maintained by the guest OS. The MMU uses only the shadow pages, which are updated after the virtualization layer validates the OS changes. The Hyper-V hypervisor, which uses shadow pages on x86, has been formally verified using

the semi automated VCC tool [17]. Related work [3, 21] uses shadow page tables to provide full virtualization, including virtual memory, for "baby VAMP", a simplified MIPS, using VCC. This work, along with later work on TLB virtualization for an abstract mode of x64 [2], has been verified using Wolfgang Paul's VCC-based simulation framework. Also, the OKL4-microvisor uses shadow paging to virtualize the memory subsystem [12]. However, this hypervisor has not been verified.

Some modern CPUs provide native hardware support for virtualization. The ARM Virtualization Extensions augment the CPU with a complete new execution mode and provide a two stage address translation. Using this mechanism, the MMU virtualization does not need to be implemented in software. Even though such hardware support can significantly reduce the complexity of the virtualization layer [24], it does not make software based solutions obsolete. For example, the recent Cortex-A5 (used in feature-phones) and the legacy ARM11 cores (used in the 2014 "New Nintendo 3DS") do not make use of such extensions. Today, the IoT and wearable computing are dominated by microcontrollers (e.g. Cortex-M). As the recent Intel Quark demonstrates, the necessity of executing legacy stacks (e.g. Linux) is pushing towards equipping these microcontrollers with a MMU. Quark and the upcoming ARMv8-R both support an MMU and lack two stage page-tables. Furthermore, solutions based on FPGAs and soft-cores (e.g. LEON) can benefit from software based virtualization since the gates that are not used for virtualization extensions can be used to implement the application specific logic (e.g. digital signal processing, software-defined radio, cryptography).

Our contributions. We present the first trustworthy virtualization mechanism based on "direct paging", an approach inspired by the paravirtualization mechanism of Xen [4]. The design of the platform is sufficiently slim to enable its formal verification without penalizing the system performance. The verification is done down to a detailed model of the architecture, including a detailed model of the ARMv7 MMU. This enable our threat model to consist of an arbitrary guest that can execute any ARMv7 instruction in user mode. We prove complete mediation of the MMU configurations, memory isolation of the hosted components, and information flow correctness. We demonstrate the platform via a prototype hypervisor that is capable of hosting a Linux system while provably isolating it from other services.

3 The Memory Virtualization API

The memory virtualization API supports two types of clients: (i) an untrusted commodity OS guest (Linux) running non-critical software (e.g. GUI, browser, server, games), and (ii) a set of trusted services such as controllers that drive physical actuators, run-time monitors, sensor drivers, or cryptographic services.

To support this use case the memory virtualization subsystem needs to provide two main functionalities:

– Isolation of memory resources used by the trusted components.

– Virtualization of the memory subsystem to enable the untrusted OS to dynamically manage its own memory hierarchy, and to enforce access restrictions.

The physical memory region allocated to each type of client is statically defined. Inside its own region the guest OS is free to manage its own memory, and the virtualization API is designed to provide the same guarantees to the guest OS as when it is running in native mode.

3.1 Memory Management

The ARMv7 MMU uses a two level translation scheme. The first level (L1) is a 4096 entry table that divides up to $4GB$ of memory into $1MB$ sections. These sections can either point to an equally large region of physical memory or to a level 2 (L2) page table with 256 entries that maps the $1MB$ section into $4KB$ physical pages.

We use direct paging [4] to virtualize the memory subsystem. Direct paging allows the guest to allocate the page tables inside its own memory and to directly manipulate them while the tables are not in active use by the MMU. Once the page tables are activated, the hypervisor must guarantee that further updates are possible only via the virtualization API to modify, allocate and free the page tables.

Physical memory is fragmented into blocks of 4 KB. Since L1 and L2 page tables have size 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. We assign a type to each physical block, that can be:

– *data*: the block can be written by the guest.
– *L1*: contains part of an L1 and is not writable in user mode.
– *L2*: contains four L2 and is not writable in user mode.

The virtualization API shown in Figure 1 is very similar to the MMU interface of the Secure Virtual Architecture [9] and consists of 9 hypercalls that selects, creates, frees, maps, or unmaps memory blocks or page tables.

3.2 Enforcing the page type constraints

Each API call needs to validate the page type, guaranteeing that page tables are write-protected. This is illustrated in Figure 2. The table in the center represents the physical memory and stores the virtualization data structures for each physical block; the page type (pt), a flag informing if the block is allocated to the guest partition (gm), and a reference counter (rc).

The four top most blocks contain an L1 page table, whose 4096 entries are depicted by the table *L1-A*. The top entry of the page table is a section descriptor ($T = S$) that grants write permission to the guest ($AP = (0, w)$). This entry points (*Add*) to the second physical section, which consists of 256 physical blocks. Three other section descriptors of the L1 are depicted in the table: the first one

switch	Select the active L1
L1create	Create page table of type L1
L2create	Create page table of type L2
L1free	Change the type of an L1 block to *data*
L2free	Change the type of an L2 block to *data*
L1unmap	Clear an entry of an L1 page table
L2unmap	Clear an entry of an L2 page table
L1map	Set an entry of an L1 page table
L2map	Set an entry of an L2 page table

Fig. 1. The virtualization API

grants write accesses to the guest, the second one gives read-only permission to the guest $(0, r)$, the third descriptor prevents any guest access and enables write permission for the privileged mode $(1, w)$. The last two entries of the L1 are PT-descriptors. These two entries point to two different L2 page tables that are stored in the same physical block.

The API calls manipulating an L1 enforce the following policy; Any section descriptor that allows the guest to access the memory must point to a section for which every physical block resides in the guest memory space. Moreover, if a descriptor enables guest to write then each block must be typed *data*. Finally, all PT-descriptors must point to physical blocks of type *L2*.

The Figure depicts two additional L1 page tables; *L1-B* and *L1-C*. The type of a physical block containing *L1-B* can be transformed to *L1* by invoking *L1create*. On the other hand, a block containing *L1-C* is rejected by *L1create* since the block contains three entries that violate the policy. In fact,

(i) the first descriptor grants guest write permission over a section which has at least one non data block, in this case *L2*,
(ii) the second section descriptor allows the guest to access a section of the physical memory in which there exists a block that is outside the guest memory, and
(iii) the third entry is a PT-descriptor, but points to a physical block that is not typed *L2*.

The table *L2-A* depicts the content of a physical block typed *L2* and that contains four L2 page tables, each consisting of 256 entries. Each hypercall that manipulates an L2 enforces the following policy: if any entry of the four L2 page tables grants access permission to the guest then the pointed block must be in the guest memory. If the entry also enables guest write access then the pointed block must be typed *data*. For example a block containing *L2-B* is rejected by *L2create*, since the block contains at least two entries that violate the policy.

A naive run-time check of the page-type policy is not efficient, since it requires to re-validate the L1 page table whenever the *switch* hypercall is invoked. To efficiently enforce that only blocks typed *data* can be written by the guest the hypervisor maintains a reference counter, which tracks for each block the sum

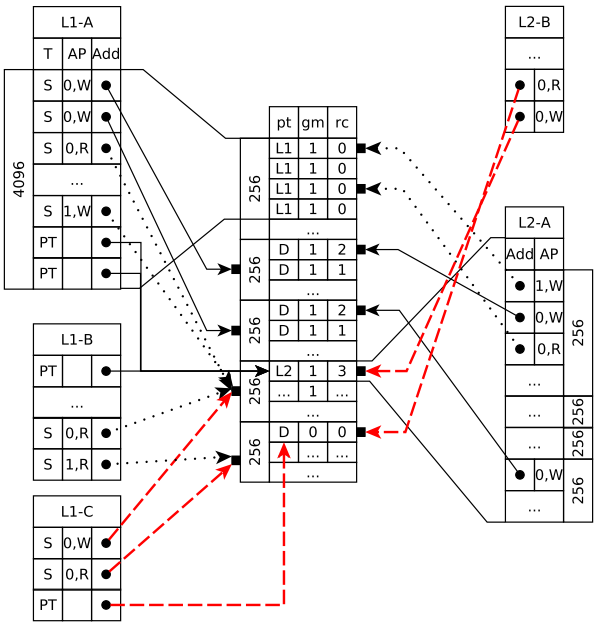

Fig. 2. Direct-paging mechanism

of (i) the number of descriptors providing writable access in user mode to the block, and (ii) the number of PT-descriptors that point to the block.

In Figure 2 we use solid arrows to represent the references that are counted and dashed arrows to represent the other references. The intuition is that a hypercall can change the type of a physical block (e.g. allocate or free a page table) only if the corresponding reference counter is zero.

3.3 Hypervisor Guest Page Table Access

The hypervisor APIs must be able to read and write guest page tables in order to check the soundness of the requests and to apply the corresponding changes. The naive solution requires handlers to change the current page table, enabling a master page table whenever the guest memory must be accessed and then re-enabling the original page table before the guest is restored. This solution is expensive as it requires to flush TLB and caches. A solution tailored for Unixes can rely on the injective mapping built by the guest, which can be used to access the guest kernel memory. However, in our setting the hosted guest is not trusted, thus this solution can not guarantee that the injective mapping is obeyed by the guest. Instead, our design reserves a subset of the virtual address space for hypervisor use. The hypervisor master page table is built so that this address space is always mapped according to an injective translation (1-to-1) allowing to easily compute the virtual address for each physical address in the

guest memory, similar to the direct memory maps supported by FreeBSD and Linux.

3.4 Memory Model and Cache Effects.

The presence of data caches and memory aliasing raise further issues. In ARMv7 CPUs such as the Cortex-A8 the MMU consults the data caches on TLB misses. When a virtual mapping is changed, the hypervisor must in general invalidate the corresponding TLB entries to guarantee that the MMU uses the updated page descriptors. However, the ARM architecture reference manual [1] predicates only weak cache coherence properties, even for single-core processors. For example, in Cortex-A8 sequential consistency is not guaranteed if the same physical address is accessed with mappings having different cacheability attributes. Thus, without knowledge of the specific processor platform, care must be taken. To ensure that the model remains valid we are forced to apply a conservative cache eviction strategy. For this reason, the hypervisor must flush the cache before accessing data stored by the guest.

More aggressive approaches (e.g. evicting only the necessary physical addresses, or avoiding flushing altogether) may be adopted for some processor implementations, but require a more fine-grained modeling including caches for their justification.

4 Verification Approach

The TLS models user mode execution of an arbitrary guest system on top of an ARMv7 CPU with MMU support, alternating with abstract handler events. These events model invocations of the hypervisor handlers as atomic transformations H_a operating on an abstract machine state. Abstract states are concrete ARMv7 states extended by auxiliary (model) data such as page types or reference counters that reflect the internal hypervisor state. Handler events represent the execution of ARMv7 instructions at privileged level, in response to exceptions or interrupts. Modeling handler effects as atomic state transformations is possible, since the hypervisor is non-preemptive, i.e. nested exceptions/interrupts are ruled out by the implementation.

4.1 TLS Consistency Properties

Since guest systems can directly manipulate inactive page tables, the TLS needs to explicitly store page tables in memory. We must show first that this does not introduce unwanted interference between guest and hypervisor state:

1. The hypervisor must act as a security monitor for the MMU settings. If *complete mediation* of the MMU settings is violated, then an attacker can bypass the access policies and compromise the security of the entire system.

2. Executions of an arbitrary guest can not affect the "trusted world" , i.e. the parts of the state the guest is not supposed to be able to write, such as non-guest memory, inaccessible processor registers and status flags, and the abstract state. We view this as an integrity property, similar to the non-exfiltration property of [13].
3. Dually, absence of information flow from the "trusted world" to the guest, confidentiality, similar to non-infiltration, must be guaranteed.

These properties, as in [13], are qualitatively different: The integrity property is first-order, and concerns the inability of the guest to directly write some other state variables. Since it is under guest control when and how to invoke the virtualization API, there are plenty of indirect communication channels connecting guests to the hypervisor. For instance, a guest decision to allocate or deallocate an L1 block affects large parts of the hypervisor state, without ever directly writing to any internal hypervisor state variable. Enforcing this is in a sense the very purpose of the hypervisor. On the other hand, the hypervisor should be unable to affect guest state even indirectly: The only desired effects of hypervisor actions should be to allocate/deallocate, map, remap, and unmap virtual memory resources, leaving any observation a guest may make unaffected. This is essentially a second-order information flow property, needed to break guest-to-guest (or guest-to-service) information channels in much the same way as intransitive noninterference is used in [19] to break guest-to-guest channels passing through the scheduler in seL4.

4.2 Refinement

Accordingly, the first verification task is to establish the model consistency properties 1, 2 and 3 above. Extending this to an actual implementation, however, requires more work, because of the TLS abstract state, and since the TLS handlers access memory using the physical addresses. The virtualization code need to execute under the same address translation as the guest, in order to minimize the number of context switches. To show implementation soundness we exhibit a refinement property relating TLS states to implementation states. We demonstrate that the refinement relation is preserved by all atomic hypervisor operations; reads and updates of the page tables, reads and updates of the hypervisor data structures. Moreover, we prove that the refinement relation directly transfers both the consistency properties and the information flow properties of the TLS to the implementation level, completing the overall memory isolation proof.

4.3 Processor Model

The verification uses the HOL4 model of ARMv7-A developed at Cambridge [11]. This model has been extensively tested and is phrased in a manner that retains a high resemblance to the pseudocode used by ARM in the architecture reference manual [1]. The Cambridge model has been extended by ourselves to include

MMU functionality. The resulting model gives a highly detailed account of the ISA level instruction semantics at the different privilege levels, including relevant MMU coprocessor effects. It must be noted that the Cambridge ARM model assumes linearizable memory, and so can be used out of the box only for processor and hypervisor implementations that satisfy this property, for instance through adequate cache flushing as discussed in section 3.

We outline the HOL4 ARMv7 model in sufficient detail to make the formal results presented later understandable. An ARMv7 machine state is a record $\sigma = \langle regs, psrs, coregs, mem \rangle \in \Sigma$, where *regs*, *psrs*, *mem* and *coregs*, respectively, represent the registers, program status registers, memory, and coprocessors. The function $mode(\sigma)$ returns the current privilege execution mode in the state σ, which can be either $PL0$ (non-privileged or user mode, used by the guest) or $PL1$ (privileged mode, used by the hypervisor). The memory is the function $mem \in 2^{32} \to 2^8$. The coprocessor registers *coregs* control the MMU.

System behavior is modeled by the state transition relation $\to_{l \in \{PL0, PL1\}} \subseteq \Sigma \times \Sigma$, where a transition is performed by the execution of an ARM instruction. Non-privileged transitions ($\sigma \to_{PL0} \sigma'$) start from and end in states that are in non-privileged execution mode (i.e. $mode(\sigma) = mode(\sigma') = PL0$). All the other transitions ($\sigma \to_{PL1} \sigma'$) involve at least one state in privileged level. The raising of an exception is modeled by a transition that enables the level $PL1$. An exception can be raised because: (i) a software interrupt (SWI) is executed, (ii) the current instruction is undefined, or (iii) a memory access is attempted that is disallowed by the MMU. Whenever an exception occurs, the CPU disables the interrupts and jumps to a predefined address in the vector table to transfer the control to the corresponding exception handler.

MMU behavior is modeled by the function $mmu(\sigma, PL, va, req)$, which takes a state σ, a privilege level, a virtual address va and an access request $req \in \{rd, wt, ex\}$ (representing read, write and execute accesses) and yields $pa \in 2^{32} \cup \{\bot\}$, where pa is the translated physical address or an access denied. The state transition relation queries the MMU whenever a virtual address is accessed, and raises an exception if the requested access mode is not allowed.

5 Formalizing the Proof Goals

A TLS state is a tuple $\langle \sigma, h \rangle$, consisting of an ARMv7 state σ and an abstract hypervisor state h of the form $\langle pgtype, pgrefs \rangle$ where *pgtype* indicates memory block types and *pgrefs* maintains reference counters. Specifically, $pgtype \in 2^{20} \to \{D, L1, L2\}$ tracks the type of each 4kb physical block; a block can either be (D) memory writable from the guest or data page, (L1) contain a L1 page table or (L2) contain a L2 page table. The map $pgrefs \in 2^{20} \to 2^{30}$ tracks the references to each physical block, as described in Section 3.

The TLS interleaves standard non-privileged transitions with abstract handler invocations. Formally, the TLS transition relation $\langle \sigma, h \rangle \to_{i \in \{0,1\}} \langle \sigma', h' \rangle$ is defined as follows:

- If $\sigma \rightarrow_{PL0} \sigma'$ then $\langle \sigma, h \rangle \rightarrow_0 \langle \sigma', h \rangle$; 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 $\sigma \rightarrow_{PL1} \sigma'$ and $mode(\sigma) = PL0$ then $\langle \sigma, h \rangle \rightarrow_1 H_a(\langle \sigma', h \rangle)$; whenever an exception is raised, the hypervisor is executed, modeled by the abstract handler H_a.

In our setup the trusted services and the untrusted guest are both executed in non-priviledged mode. To distinguish between these two partitions, we use ARM domains. In the ARM architecture domains are the primarily access control mechanism used by the MMU. This mechanism is orthogonal to the CPU execution modes. The architecture provides sixteen domains, each of them can be activated independently. We reserve the domains 2-15 for the secure services. In the following we use the predicate $S(\sigma)$ to identify if the active partition is the one hosting the secure services: the predicate holds if at least one of the reserved domain is enabled.

5.1 TLS Consistency

We introduce a system invariant $I(\langle \sigma, h \rangle)$ used to constrain the set of consistent initial states of the TLS. The invariant is needed, for instance, to ensure that guests have write access to page tables only when they are inactive. We use $\mathcal{Q}_I$ to represent the set of all possible TLS states that satisfy the invariant. We thus need to show:

Theorem 1. *Let* $\langle \sigma, h \rangle \in \mathcal{Q}_I$ *and* $i \in \{0, 1\}$. *If* $\langle \sigma, h \rangle \rightarrow_i \langle \sigma', h' \rangle$ *then* $I(\langle \sigma', h' \rangle)$.

We say that two states are *MMU-equivalent* if for any virtual address va the MMU yields the same translation and the same access permissions. Formally, $\sigma \equiv_{mmu} \sigma'$ if and only if

$$mmu(\sigma, PL, va, req) = mmu(\sigma', PL, va, req)$$

for any va, PL, req. Complete mediation (MMU-integrity) is demonstrated by showing that neither the guest nor the secure services are able to directly change the content of the page tables and affect the address translation mechanism.

Theorem 2. *Let* $\langle \sigma, h \rangle \in \mathcal{Q}_I$. *If* $\langle \sigma, h \rangle \rightarrow_0 \langle \sigma', h' \rangle$ *then* $\sigma \equiv_{mmu} \sigma'$.

We use the approach of [13] to analyze the hypervisor data separation properties. The observations of the guest in a state $\langle \sigma, h \rangle$ is represented by the structure $O_g(\langle \sigma, h \rangle) = \langle uregs, cpsr, mem_g, coregs \rangle$ of user registers $uregs$, control register $cpsr$, guest memory mem_g and coprocessor registers $coregs$. The register $cpsr$ and the coprocessor registers are visible to the guest since they directly affect guest behavior, and do not contain any information the guest should not be allowed to see. Evidently, however, all writes to the coprocessor registers must be mediated by the hypervisor.

The remaining part of the state (i.e. the content of the memory locations that are not part of the guest memory, special registers) and, again, the coprocessor

registers constitute the secure observations $O_s(\langle\sigma, h\rangle)$ of the state, which guest transitions are not supposed to affect.

The following theorem demonstrates that the context switch between the untrusted guest and the trusted services is not possible without the mediation of the hypervisor. The proof is straightforward, since S only depends on coprocessor registers that are not modifiable in nonprivileged mode.

Theorem 3. *Let* $\langle\sigma, h\rangle \in \mathcal{Q}_I$. *If* $\langle\sigma, h\rangle \rightarrow_0 \langle\sigma', h'\rangle$ *then* $S(\sigma) = S(\sigma')$.

The *non-exfiltration* property guarantees that a transition executed by the guest does not modify the secure resources:

Theorem 4. *Let* $\langle\sigma, h\rangle \in \mathcal{Q}_I$. *If* $\langle\sigma, h\rangle \rightarrow_0 \langle\sigma', h'\rangle$ *and* $\neg S(\sigma)$ *then* $O_s(\langle\sigma, h\rangle) = O_s(\langle\sigma', h'\rangle)$.

The *non-infiltration* property is a non-interference property guaranteeing that guest instructions and hypercalls executed on behalf of the guest do not depend on any information stored in resources not accessible by the guest.

Theorem 5. *Let* $\langle\sigma_1, h_1\rangle, \langle\sigma_2, h_2\rangle \in \mathcal{Q}_I$, $i \in \{0, 1\}$, *and assume that* $O_g(\langle\sigma_1, h_1\rangle) = O_g(\langle\sigma_2, h_2\rangle)$, $\neg S(\sigma_1)$ *and* $\neg S(\sigma_2)$. *If* $\langle\sigma_1, h_1\rangle \rightarrow_i \langle\sigma_1', h_1'\rangle$ *and* $\langle\sigma_2, h_2\rangle \rightarrow_i \langle\sigma_2', h_2'\rangle$ *then* $O_g(\langle\sigma_1', h_1'\rangle) = O_g(\langle\sigma_2', h_2'\rangle)$.

5.2 The Implementation Model

A critical problem of verifying low level platforms is that intermediate states of the MMU configuration can break the semantics of the high level language (e.g. C). This is the reason we introduced the implementation model, that is sufficiently detailed to expose misbehavior of the hypervisor accesses to the observable part of the memory (i.e. page tables, guest memory and internal data structure). The implementation interleaves standard non-privileged transitions and hypervisor functionalities. In contrast to the TLS, these functionalities now store their internal data in system memory, accessed by means of virtual addresses. In practice, in the implementation model the hypervisor functionalities are expressed as executable specifications that are, however, very to close the actions executed by an actual machine at instruction semantics level. We demonstrate these differences by comparing two fragments of the TLS and the implementation specifications.

The TLS models the update of a guest page table descriptor as $\sigma'.mem = write_{32}(\sigma.mem, pa, desc)$, where pa is the physical address of the entry, $desc$ is a word representing the new descriptor and $write_{32}$ is a function that yields a new memory having four consecutive bytes updated. At the implementation level the same operation is represented as

```
if ¬ mmu(σ, PL1, Gpa2va(pa)).wt
 then ⊥
 else write32(σ.mem,mmu(σ,PL1,Gpa2va(pa)).pa,desc)
```

where `Gpa2va` is the function used by the hypervisor to compute the virtual address of a physical address that resides in guest memory. This function is statically defined and is the inverse of the injective translation established by the hypervisor master page table. The implementation can fail to match the TLS for two reasons: (i) the current page table can prevent the hypervisor from accessing the computed virtual address, and then the implementation terminates in a failing state (denoted by $\perp$), (ii) the current address translation does not respect the expected injective mapping, thus $mmu(\sigma, PL1, Gpa2va(pa)).pa \neq pa$ and the implementation writes in an address that differs from the one updated by the TLS.

The next example shows the difference between access of the reference counter in the TLS and at implementation level. The TLS models this operation as $h.refs(b)$, where b is the physical block. The implementation models the same operation using memory offsets as follows:

```
if ¬ mmu(σ, PL1, tbl_va + 4*b).rd )
 then ⊥
 else read_32(σ.mem,mmu(σ, PL1, tbl_va + 4*b).pa) & 0xCFFFFFFF
```

This representation is directly reflected in the hypervisor code. For each block, the page type (two bits) and the reference counter (30 bits) are placed contiguously in a word. These words form an array, whose initial virtual address is tbl_{va}.

The concrete handlers are represented by a HOL4 function H_r from concrete ARMv7 states to concrete ARMv7 states. The function is the executable specification of the various exception handlers including the MMU functionalities.

Then, implementation behavior is determined by the state transition relation $\rightarrow_{i\in\{0,1\}} \subseteq \Sigma \times (\Sigma \cup \{\perp\})$ as follows:

- If $\sigma \rightarrow_{PL0} \sigma'$ then $\sigma \rightarrow_0 \sigma'$; instructions executed in non-privileged mode that do not raise exceptions behave according to the standard ARMv7 semantics.
- If $\sigma \rightarrow_{PL1} \sigma'$ and $mode(\sigma) = PL0$ then $\sigma \rightarrow_1 H_r(\sigma')$; whenever an exception is raised, the hypervisor is executed and its behavior is modeled by the function H_r.

5.3 The Refinement

To show implementation soundness we exhibit a refinement property relating abstract states $\langle \sigma_1, h \rangle$ to concrete states σ_2. The refinement relation $\mathcal{R}$ requires that: (i) the registers and coprocessors contain the same value in both states, (ii) the guest memory contains the same values in both states, (iii) part of the memory of the implementation state contains a mapping of the hypervisor data structures of the TLS state and (iv) the reserved virtual addresses are always mapped equivalently to the master page table. Observations of the guest O_g are defined on concrete states using the hypervisor data structure mapping in analogy with the corresponding observations on abstract states defined above.

Theorem 6. *Let* $\langle \sigma_1, h \rangle \in \mathcal{Q}_I$ *and* $\sigma_2 \in \Sigma$ *such that* $\langle \sigma_1, h \rangle \mathcal{R} \sigma_2$. *Let* $i \in \{0, 1\}$. *Then* $\sigma_2 \rightarrow_i \sigma_2'$ *if and only* $\langle \sigma_1, h \rangle \rightarrow_i \langle \sigma_1', h' \rangle$ *and* $\langle \sigma_1', h' \rangle \mathcal{R} \sigma_2'$.

Finally we show that the security property of the TLS and the refinement relation directly transfer the mmu-integrity/non-exfiltration/non-infiltration to the implementation. We use Σ_I to represent the space of consistent concrete states: States σ_2 such that if $\langle \sigma_1, h \rangle \mathcal{R} \sigma_2$ then $I(\langle \sigma_1, h \rangle)$.

Corollary 1. *Let* $\sigma_1, \sigma_2 \in \Sigma_I$, $i \in \{0, 1\}$ $O_g(\sigma_1) = O_g(\sigma_2)$*:*

- *if* $\sigma_1 \rightarrow_0 \sigma_1'$ *then* $\sigma_1 \equiv_{mmu} \sigma_1'$
- *if* $\sigma_1 \rightarrow_0 \sigma_1'$ *and* $\neg S(\sigma_1)$ *then* $O_s(\sigma_1) = O_s(\sigma_1')$
- *if* $\sigma_1 \rightarrow_i \sigma_1'$, $\sigma_2 \rightarrow_i \sigma_2'$, *and* $\neg S(\sigma_1)$ *and* $\neg S(\sigma_2)$ *then* $O_g(\sigma_1') = O_g(\sigma_2')$

6 Linux support

To evaluate the real-world feasibility of our approach we examine a virtualized Linux guest. The Linux kernel v2.6.34 has been modified to run on top of the hypervisor. This task required modification of architecture-dependent parts of the Linux kernel like (i) execution modes, (ii) low-level exception routines and (iii) page table management. High-level OS functions such as process, resource and memory manager, file system and networking did not require any modifications.

CPU privilege modes. The target CPU includes only two execution modes: privileged and unprivileged (user). Like for other approaches based on paravirtualization, since the hypervisor executes as privileged, then the Linux kernel has been modified to execute as unprivileged. To separate kernel and user applications, the hypervisor manages two separate unprivileged execution contexts: virtual user and virtual kernel modes. In x86 these virtual modes can be implemented by segmentation limits. This approach is not possible for CPUs that do not provide this feature (e.g. x86 64-bits and ARM). Instead, for kernel-user space isolation we use ARM domains, that implement an access control regime orthogonal to the CPU execution modes. Notice that the main security goal here is not to guarantee this OS-internal isolation, but to maintain the separation between the virtualized components.

CPU exceptions. CPU exceptions such as aborts and interrupts change the processor mode to privileged. These exceptions must therefore be handled in the hypervisor, which after validation can forward them to the unprivileged exception handlers of the Linux kernel. The hypervisor supplies the kernel exception handlers with some privileged data needed to correctly service an on-going exception (i.e. for pre-fetch abort, the privileged fault address and fault status registers are forwarded to the guest). The exception handlers in the Linux kernel have thus been slightly modified to support this.

Memory management. Within the Linux kernel, virtual memory is handled in two layers. The first is platform independent and provides a number of high-level functions to the rest of the kernel. The second layer provides a number of platform dependent functions to the first layer. To allow virtualization, we modified the second layer to perform a hypercall instead of performing privileged access to the hardware.

7 Benchmark and evaluation

Runtime overhead. To analyze runtime overhead we use LMBench [18] (of which the fork benchmarks stress the MMU virtualization) running on Linux 2.6.34 [2] with and without virtualization. The outcome, measured on an ARMv7-A Cortex-A8 powered embedded system (BeagleBoard-xM), is presented in Table 1. Additionaly, we use the creation (tar) and compression (gzip) of archives as macrobenchmarks. The significant virtualization overhead for the fork benchmarks is due to a large number of simple operations (in this case, write access to a page-table) being replaced with a large number of expensive hypercalls. It may be possible to reduce this overhead with minimal optimization (e.g. batching).

In Table 1 we also report the overheads measured in [15] of several existing hypervisors for ARM. We point out that these performance numbers have been obtained from different sources, testing different ARM cores, boards and hosted Linux kernels. Moreover, the numbers presented here use a completely unoptimized version of the hypervisor that we believe can be significantly improved.

Benchmark	Our Hypervisor	L4Linux	Xen	OKL4
null syscall	342%	2955%	150%	60%
read	155%	836%	90%	15%
write	181%	874%	84%	24%
stat	93%	553%		60%
open/close	146%	433%		-10%
select (10)	41%	363%		14%
pipe	115%	449%	73%	31%
fork+exit	164%	949%	246%	8%
fork+execve	166%	591%	238%	5%
fork+bin/sh-c	60%	415%		
targz 500KB	1%			
targz 1MB	-4%			
targz 2MB	-1%			

Latency benchmarks

Processes	DP 256MB	DP 1GB	SPT
32	56	224	608
64	64	256	1216
128	72	288	2432

Memory usage (KB) of direct paging (mem_{size}) and shadow page tables

Table 1. Benchmarks

Footprint The main difference between our proposal and the existing verified hypervisors is the MMU virtualization mechanism. The direct paging approach requires a table which contains at most $mem_{size}/block_{size}$ entries, where mem_{size} is the total available physical memory and $block_{size}$ is the minimum page size (here, $4KB$). Each entry in this table uses $2 + \log_2\ max_{ref}$ bits, with the first two bits used to record entry type and max_{ref} being the maximum number of references pointing to the same page. Assuming this number is bound by the number of processes, Table 1 indicates the memory overhead introduced by direct paging. It should be noted that on ARMv7, most operating systems including Linux dedicate one L1 page to each process and at least three L2 pages to map the stack, the executable code and the heap. Then the OS itself has a

[2] The virtualization API is independent of the hosted OS, thus porting and running a different Linux kernel or BSD does not affect the security properties described in this paper

minimum footprint of $16KB + 3 * 1KB$ per process. This footprint is doubled if the underlying hypervisor uses shadow page tables.

Implementation and verification effort The hypervisor is implemented in C (and some assembly) and consists of 4529 lines of code (LOC). Excluding platform dependent parts, the hypervisor core is no larger than 2066 LOC. The memory virtualization subsystem consists of 1200 LOC. To paravirtualize Linux we changed 1025 LOC of its kernel, 950 in the ARM specific architecture folder and 75 in init/main.c. The paravirtuation is binary compatible with existing userland applications. For comparison, the only other hypervisor that implements direct paging is the Xen hypervisor, which consists of 100KLOC and its design is not suitable for verification. Instead, the small code base of our hypervisor makes it easier to experiment with different virtualization paradigms and enables formal verification of its correctness. The formal specification consists of 1500 LOC of HOL4 and intentionally avoids any high level construct, in order to make the model as similar as possible to the implementation, at the price of increasing the verification cost. The proof consists of 18700 LOC of HOL4.

The verification highlighted a number of bugs in the initial design of the APIs: (i) arithmetic overflow when updating the reference counter, caused by not preventing the guest to create an unbounded number of references to a physical block, (ii) bit field and offset mismatch, (iii) missing check that a newly allocated page table prevents the guest to overwrite the page table itself, (iv) usage of signed shift operator where the unsigned one was necessary and (v) approval of guest requests that cause unpredictable MMU behavior. The verification of the implementation model identified three additional bugs exploitable by the guest by requesting the validation of page tables outside the guest memory.

The project was conducted in three steps. The design, modeling and verification of the APIs for memory virtualization required nine person months. Here, the most expensive tasks have been the verification of Theorems 1 [20] and 6. The C implementation of the APIs and the Linux port has been accomplished in three months. While the implementation team was completing the Linux port the verification team started the verification of the refinement, which has taken three months so far. This work is continuing, in order to complete the verification from the HOL4 implementation level down to assembly.

8 Concluding remarks

We presented the first hypervisor (i) for a COTS application processor architecture (ARMv7), (ii) whose spatial separation properties have been formally verified, (iii) capable of hosting a Linux system. As example application, in [6] we used the virtualization mechanism to support a tamper-proof run-time monitor that prevents code injection in an untrusted Linux guest.

The only verified hypervisor in the literature capable of hosting a commodity OS is Microsoft's Hyper-V [17]. However, little detailed information about the Hyper-V internal structure or the Hyper-V verification exercise is publicly available. As part of the Hyper-V verification project, a hypervisor for a simplified,

MIPS-like architecture including memory virtualization is described in [3, 21]. However, the relation of the simplified hypervisor to Hyper-V itself is not clear. As other, unverified, hypervisors for ARM such as the OKL4 microvisor [12] the Hyper-V precursor of Paul et al uses shadow page tables for MMU virtualization. Our result demonstrates that secure isolation of a commodity OS can be achieved with highly promising performance without requiring either specialized hardware support or shadow data structures. This applies even before assembly level and cache related optimizations are performed. This represents the first trustworthy virtualization mechanism based on "direct paging", an approach inspired by the paravirtualization mechanism of Xen.

The implementation model takes into account low-level details (i.e. virtual address translation, bit field manipulation, finite integer arithmetic, accesses to the hypervisor data not mediated by high level data structures) and represent an executable specification. The model is sufficiently detailed to spot possible errors that arise when the hypervisor uses virtual addresses and exactly reflects the control flow of the C-implementation. Part of our ongoing research efforts is to adapt existing techniques [23] to verify the hypervisor binary code.

References

1. *ARMv7-AR Architecture Reference Manual*. Technical documentation ARM DDI 0406B. ARM Limited, 2008.
2. E. Alkassar, E. Cohen, M. Kovalev, and W. J. Paul. Verification of tlb virtualization implemented in c. In *Verified Software: Theories, Tools, Experiments*, pages 209–224. Springer, 2012.
3. E. Alkassar, M. A. Hillebrand, W. Paul, and E. Petrova. Automated verification of a small hypervisor. In *Verified Software: Theories, Tools, Experiments*, pages 40–54. Springer, 2010.
4. P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.
5. X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 2–13. ACM, 2008.
6. H. Chfouka, H. Nemati, R. Guanciale, M. Dam, and P. Ekdahl. Trustworthy prevention of code injection in linux on embedded devices. In *ESORICS 2015: To appear*. 2015.
7. J. Criswell, N. Dautenhahn, and V. Adve. Kcofi: Complete control-flow integrity for commodity operating system kernels. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 292–307. IEEE, 2014.
8. J. Criswell, N. Dautenhahn, and V. Adve. Virtual ghost: Protecting applications from hostile operating systems. In *SIGARCH Computer Architecture News*, pages 81–96. ACM, 2014.
9. J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 351–366. ACM, 2007.

10. M. Dam, R. Guanciale, N. Khakpour, H. Nemati, and O. Schwarz. Formal verification of information flow security for a simple arm-based separation kernel. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 223–234. ACM, 2013.
11. A. C. J. Fox and M. O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *proc. ITP'10*, volume 6172 of *Lecture Notes in Computer Science*, pages 243–258. Springer, 2010.
12. G. Heiser and B. Leslie. The okl4 microvisor: Convergence point of microkernels and hypervisors. In *Proceedings of the first ACM asia-pacific workshop on Workshop on systems*, pages 19–24. ACM, 2010.
13. C. Heitmeyer, M. Archer, E. Leonard, and J. McLean. Applying formal methods to a certifiably secure software system. *IEEE Trans. Softw. Eng.*, 34(1):82–98, Jan. 2008.
14. O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. Inktag: secure applications on an untrusted operating system. *ACM SIGPLAN Notices*, 48(4):265–278, 2013.
15. A. Iqbal, N. Sadeque, and R. I. Mutia. An overview of microkernel, hypervisor and microvisor virtualization approaches for embedded systems. *Report, Department of Electrical and Information Technology, Lund University, Sweden*, 2110, 2009.
16. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *Proc. SOSP'09*, pages 207–220. ACM, 2009.
17. D. Leinenbach and T. Santen. Verifying the Microsoft Hyper-V hypervisor with VCC. In *Proc. FM'09*, volume 5850 of *Lecture Notes in Computer Science*, pages 806–809. Springer Berlin Heidelberg, 2009.
18. L. McVoy and C. Staelin. Lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association.
19. T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. sel4: from general purpose to a proof of information flow enforcement. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 415–429. IEEE, 2013.
20. H. Nemati, R. Guanciale, and M. Dam. Trustworthy virtualization of the armv7 memory subsystem. In *SOFSEM 2015: Theory and Practice of Computer Science*, pages 578–589. Springer, 2015.
21. W. Paul, S. Schmaltz, and A. Shadrin. Completing the automated verification of a small hypervisor–assembler code verification. In *Software Engineering and Formal Methods*, pages 188–202. Springer, 2012.
22. R. Richards. Modeling and security analysis of a commercial real-time operating system kernel. In D. S. Hardin, editor, *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pages 301–322. Springer US, 2010.
23. T. A. L. Sewell, M. O. Myreen, and G. Klein. Translation validation for a verified os kernel. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 471–482. ACM, 2013.
24. P. Varanasi and G. Heiser. Hardware-supported virtualization on arm. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 11:1–11:5, New York, NY, USA, 2011. ACM.