Formal Verification of Information Flow Security for a Simple ARM-Based Separation Kernel

Mads Dam, Roberto Guanciale,
Narges Khakpour, Hamed Nemati
KTH Royal Institute of Technology
SE-100 44, Stockholm, Sweden
{mfd, robertog, nargeskh,
hnnemati}@kth.se

Oliver Schwarz
SICS Swedish ICT
Box 1263
SE-164 29
Kista, Sweden
oliver@sics.se

ABSTRACT

A separation kernel simulates a distributed environment using a single physical machine by executing partitions in isolation and appropriately controlling communication among them. We present a formal verification of information flow security for a simple separation kernel for ARMv7. Previous work on information flow kernel security leaves communication to be handled by model-external means, and cannot be used to draw conclusions when there is explicit interaction between partitions. We propose a different approach where communication between partitions is made explicit and the information flow is analyzed in the presence of such a channel. Limiting the kernel functionality as much as meaningfully possible, we accomplish a detailed analysis and verification of the system, proving its correctness at the level of the ARMv7 assembly. As a sanity check we show how the security condition is reduced to noninterference in the special case where no communication takes place. The verification is done in HOL4 taking the Cambridge model of ARM as basis, transferring verification tasks on the actual assembly code to an adaptation of the BAP binary analysis tool developed at CMU.

Categories and Subject Descriptors

D4.6 [**Operating Systems**]: Security and Protection; D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods, correctness proofs*

Keywords

Formal verification; Information Flow Security; Separation Kernel; Hypervisor

1. INTRODUCTION

The design of secure systems needs to ensure that software components belonging to different security domains are adequately isolated from each other, such that only authorized communication can take place between them. One way of achieving this is by dedicated hardware, e.g. TPM's or SIM's. This, however, carries significant overhead, in terms of the hardware itself, and the associated infrastructure. An alternative is to execute the components in isolated partitions on shared hardware, using low-level software execution platforms such as separation kernels [25, 12] or secure hypervisors [10, 27, 17]. A key requirement of this solution is the verification of the tamper resistant trusted computing base, preferably by use of formal methods. Significant progress has been made recently in this direction, cf. the seL4 project [14], Microsoft's Hyper-V project [16], and Green Hills' CC certified INTEGRITY-178B separation kernel [23].

Our focusing scenario consists of an "untrusted" component (e.g. a smartphone software stack) that interacts with a set of trusted services, such as a virtual SIM card, or a virtual TPM. A shared execution platform for this scenario needs to provide the following minimal functionality:

1. Isolation of component resources
2. A communication mechanism that plays the role of external communication lines in a physically distributed system
3. A scheduling mechanism to commit shared resources (e.g. processors) among the components.

In this paper we present a proof-of-concept design and machine level verification of the PROSPER separation kernel for ARMv7 [3], which supports the above functionality. The kernel allows the execution of two component systems, such as a smartphone OS with a virtualized SIM application, on top of a single physical machine. The interesting feature of this set-up is that explicit, kernel-supported communication between partitions is essential, and a critical aspect of the security analysis is to ensure that this communication does not introduce (deliberate or accidental) side channels that can be exploited by an attacker.

This security analysis is far from trivial. The objective of a separation kernel[1], following Rushby [25], is first to make

CCS'13, November 4–8, 2013, Berlin, Germany.
ACM 978-1-4503-2477-9/13/11.
http://dx.doi.org/10.1145/2508859.2516702.

[1] We use the label "separation kernel" in this paper mainly since no support for user/kernel space virtualization is provided to the component systems, but the borderline between separation kernels and secure hypervisors, and our use of the associated terminology, is admittedly fuzzy.

it appear that each component system is executed on a separate, isolated, machine, and second to ensure that communication can only flow as authorized along known external channels. However, there are several problems in delegating communication to an external agent. First, it entails an extension of the trusted computing base to include the external channel itself. Second, virtualizing the component systems without also virtualizing the channel connecting them is hardly a reasonable design. Third, potential side channels are ignored. In a case such as this, where a partition must be able to access the virtualized SIM application at will, communication can convey critical timing information that an attacker can exploit to extract key material, as is well known [15].

We propose instead an approach where communication between partitions is made explicit in the top level specification (TLS), and information flow is analyzed in the presence of such an intended communication channel. We formulate the TLS such that it directly formalizes, in sufficient detail, the set of computation paths both allowed and required at the implementation level, and then we check that the implementation indeed satisfies this specification. The question is how to do this, if it can be done while maintaining a satisfactory account of isolation, and if it can be done at a satisfactory level of abstraction (such that the TLS does not conflate to become identical to the implementation itself). In this paper we present a proof-of-concept solution in the sense that functionality is limited as much as meaningfully possible, but such that the specification, implementation and correctness proof is carried through in complete detail from TLS to realization for ARMv7, and proved correct at the instruction reference semantics level using the HOL4 model of ARM developed at Cambridge [7].

In our case, the goal of verification is to show that there is no way for the partitions to affect each other, directly or indirectly, *except* through the intended channel. In particular, there should be no way for a partition to access the memory or register contents, by reading or writing, of the other partition, other than when the communication is realized by explicit usage of the intended channel, by both partitions in collaboration. This is not an easy property to reconcile with the standard information flow tools such as noninterference (NI) or intransitive noninterference:

- NI is problematic since the very purpose of the kernel is to *allow* rather than prevent information flow (through the intended channel). For the sake of illustration consider the recent NI-based verification of the seL4 kernel [20, 19]: A critical step in that work boils down to a proof of the *absence* of information flow from the previously scheduled partition to the scheduler itself, in order to prevent the scheduler being used as a communication channel. This type of approach is not applicable in our setting since communication must be allowed to affect the partitions in ways that are outside our control, as the partitions are not statically known. Moreover, for the same reason we have no control over what, where, or how the channel is intended to be used, and thus the various NI-based declassification schemes (cf. [26]) do not help.

- Intransitive noninterference [24] relaxes NI with the possibility to add unknown, but trusted, intermediary agents through which information flows can be required to pass. In our case that agent is the kernel, which is known, and not a priori trusted. This makes intransitive noninterference difficult to make use of in the present context.

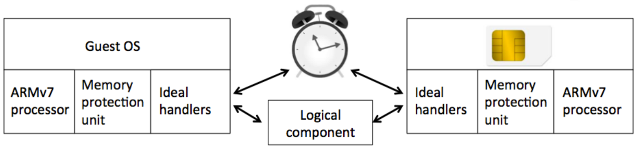

Figure 1: Ideal model

We thus take a different approach. We formulate the TLS as an "ideal model" which satisfies the required separation properties by construction, and then reduce correctness to trace equivalence w.r.t. a "real model", reflecting actual systems behaviour. The key idea is to execute the partitions on physically separate, ideal processors, connected by an explicit, ideal communication channel, and equipped with a little extra paraphernalia, as shown in Fig. 1. The ideal processors need to accurately mimick the execution of user space partitions on a real processor. This is done by augmenting the TLS processors by idealized functionality, the "ideal handlers" of Fig. 1, which is invoked whenever the actual processors would transition to privileged mode by an exception (e.g. a hardware interrupt, or an exception). This construction allows userland code to execute as desired (with the exception of fine grained timing differences we currently do not take into account), but the idealized processors are physically prevented from directly affecting their sibling machine, with the exception of explicit communication using the message delivery service.

The task is thus to show that, properly set up, the user observable traces of the ideal model are the same as those of a "real model", obtained by executing the software in different partitions on top of our separation kernel, on top of a real ARMv7-A processor, including a Memory Management Unit (MMU) for physical protection of memory regions belonging to different partitions. We prove this using the bisimulation proof method [28], by exhibiting a concrete bisimulation relation, a *candidate relation*, relating the state spaces of ideal and real models. The proof that the candidate relation is actually a bisimulation relation of the appropriate type is in turn reduced to subsidiary properties, several of which have natural correspondences in previous kernel verification literature, cf. [12, 23], namely that:

i The initial states of the ideal and the real models are in the candidate relation. This is ensured by the correctness of the bootstrap code.

ii A partition does not perform any isolation-violating operation while it is executing. Due to our use of memory protection, this is really a noninterference-like property of the ARMv7-A architecture rather than a property of the separation kernel. This property is similar to the partition management result reported in [30].

iii The processor state switches correctly upon transition to privileged mode. Again this is a processor architecture rather than a kernel dependent result.

iv Execution of ideal exception handlers vs the real separation kernel exception handlers preserve the candidate relation.

v Several invariants are preserved while partitions and /or kernel handlers are running.

For the actual verification we have used a combination of theorem proving and binary code analysis. The real and ideal models are built on top of the Cambridge ARM HOL4 model, extended with a simple MMU unit. The isolation lemmas of ARM, items (ii) and (iii), are proved using a tool, ARM-prover, developed for the purpose in HOL4. The proofs are costly and involve traversing the full ARM instruction sets. The ARM-prover allows the proofs to be automated to a large extent. This frees us from the onerous task of verifying the two theorems on each element of the large ARM instruction set. The ARM-prover tool is developed on top of the monadic ARM semantics reported in [7]. Handlers (item (iv)) are verified using pre/post conditions. Manual generation of these handlers is an error-prone process, and for this reason we generate the pre/post conditions automatically based on the specification of the ideal model, the candidate relation, and the ARM isolation Lemmas. We transfer the pre/post conditions to the binary code analysis tool BAP [6], and use BAP to verify the bootstrap code and the kernel handlers at the assembly code level. Several tools have been developed for lifting the ARM code to the input format of BAP and manipulating the code. Space prevents us from more than outlining the ARM isolation lemmas and the BAP extensions here; these will be reported in separate publications.

The paper is organized as follows: In section 2 we present the ARMv7 processor, MMU, and timing model. In section 3, the PROSPER kernel is presented, and the the ideal model is described in section 4. We then proceed to give an overview of the proof strategy, including the decomposition of the TLS into the ARM lemmas, and the handler verification tasks. In section 6 we partially validate our verification approach by proving a "monotonicity of release" property as suggested by Sabelfeld and Sands [26], that a corollary of our proof is an NI property in the special case where partitions do not actually communicate. In section 7 we present the proof implementation, and in section 8 information is given on the status of the kernel and some performance figures regarding the proof itself. In section 9 related work is discussed, and finally in section 10, we conclude and discuss some unresolved issues.

2. ARMv7

An ARMv7 CPU has execution mode $m \in \mathcal{M}$ where $\mathcal{M} = \{usr, svc, abort, undef, irq, fiq, sys\}$. The non-privileged *usr* mode is used by the user partitions, while the privileged modes $\mathcal{M}_p = \mathcal{M} \backslash \{usr\}$ are used to execute kernel activities. A machine state is a record $\sigma = \langle regs, psrs, mem, coregs \rangle$ where *regs*, *psrs*, *mem* and *coregs*, respectively represent the registers, program status registers (*psrs*), memory and coprocessors of the machine. The register set *regs* consists of the sixteen user registers that are accessible in all modes as well as the banked-registers of each privileged mode that are available only in that mode. The program status registers is a record

$$\langle cpsr, psr_{svc}, psr_{abort}, psr_{undef}, psr_{irq}, psr_{fiq}, psr_{sys} \rangle$$

where *cpsr* is the current psr and each psr_m is the banked psr in mode $m \in \mathcal{M}_p$. A psr encodes the arithmetical flags, the executing mode, the interrupt mask, and the instruction decoding. The functions $I(\sigma)$ and $M(\sigma)$ return the hardware interrupt mask and the current mode in state σ, respectively. Moreover, the memory is the function $mem \in word32 \rightarrow word8$.

The tuple $coregs = \langle c_1, c_2, c_3 \rangle$ contains the three 32-bit registers of coprocessor CP15, used mainly to control the Memory Management Unit (MMU). The register c_1 represents whether the MMU is enabled or not, and c_2 gives the base address of the page table. In ARMv7, there are sixteen domains, each representing a security role. The coprocessor register c_3 holds the current status of the domains. An entry of the page table determines the owner domain of the corresponding page and its access permission.

In our setting, a "real' system" is an ARM machine connected to a timer device. A real system is modeled by the record $s = \langle \sigma, t \rangle \in \mathcal{S}$, where σ is an ARM machine state and t represents the clock cycles elapsed since system start. The behavior of a system is defined by the state transition relation $\rightarrow \subseteq \mathcal{S} \times \mathcal{S}$ where a transition is performed due to either the execution of an ARM instruction or a timer signal. We assume a simple time model that constrains all transitions to consume one clock cycle, i.e. if $\langle \sigma, t \rangle \rightarrow \langle \sigma', t' \rangle$ then $t' = t + 1$.

If the real system switches from the mode *usr* to a privileged mode, then an exception has occurred. The privileged mode *svc* is activated by a software interrupt (SWI). If the current instruction is undefined, then the system switches to the mode *undef*. The mode *irq* is activated by a hardware interrupt. In our setting, the timer triggers a hardware interrupt every fixed amount of clock (actually, instruction) cycles. If the MMU prevents an access to the memory, then the mode *abort* is enabled. In our setting, no exception can activate the modes *sys* and *fiq*. In fact, *sys* mode can only be explicitly entered from a privileged mode. Moreover, in our model there is only one device (the timer) which delivers standard hardware interrupts. For this reason the fast interrupt mode *fiq* is never activated. Whenever an exception occurs, the CPU backs up the program counter and the cpsr into the banked registers and into the psr of the activated mode, disables hardware interrupts and jumps to a predefined address in the vector table to transfer the control to the corresponding exception handler.

Figure 2 (A) depicts an example computation of a real system. White and grey circles represent states in user mode and black circles represent states in privileged modes. The circle labels represent the system clocks and the solid arrows represent the transition relation. Transitions between two states in user mode (e.g. 1 → 2) do not cause any exceptions. The timer tick of this example is six clock cycles, then an interrupt is delivered in the states 6 and 12, switching the system to mode *irq*. The transition between the states 2 and 3 is caused by a different exception, for example the execution of a software interrupt. Finally, transitions from privileged modes to user mode (e.g. 4 → 5) are caused by instructions that explicitly change *cpsr*.

3. THE PROSPER KERNEL

The PROSPER kernel has four minimal functionalities: execution of two user partitions on one physical machine, protection of the partition resources, partition scheduling,

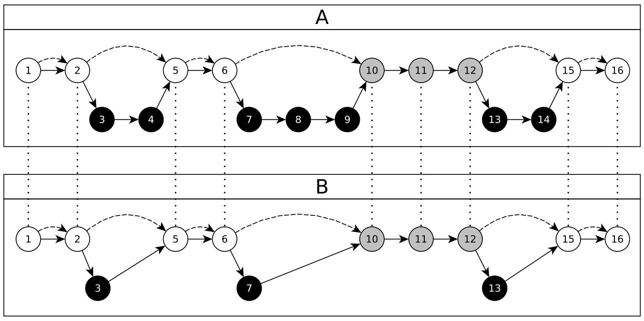

Figure 2: (A) The real world and (B) the Top Level Specification

and inter-partition communication. All low-level tasks of the kernel that depend on the architecture (e.g. accessing special registers and coprocessors, context saving and restoring) are implemented in assembly ($\sim$ 150 lines of codes), while all high-level tasks (e.g., hypercall, scheduling, page table setup) are implemented in C ($\sim$ 600 lines of code). The current implementation can host OSs (e.g. μClinux [18], FreeRTOS[8]) that do not require intra-partition memory protection.

The machine memory is partitioned into three separate regions: the region in the range of $[min_g, max_g]$ for the partition $g \in \{1, 2\}$, and a kernel memory region. The accesses to the partitions are controlled by the MMU where three ARMv7 domains 0,1, and 2 are used to represent the kernel, the first partition and the second partition, respectively. When the kernel resumes a partition, it updates the coprocessor c_3 to set the partition domain to the value *client* and to disable the domain of the other partition. The MMU is configured to enforce the following properties: (i) if a partition is running, then only its memory can be accessed, (ii) whenever the kernel is activated (e.g. a partition performs a software interrupt), it is able to read and write its memory and the memory of the "interrupted" partition. We have no concurrency inside the kernel, i.e. an exception can not interrupt while another exception is being handled.

The partitions communicate through asynchronous message passing. Each partition has two executing status variables: *message status* is intended to process the incoming messages while the *task status* is used for other activities. To each status is associated a context that contains a set of user registers and the cpsr. For the active partition, the context corresponding to the active status is the current user registers and the cpsr and the context of the non-active status is stored in the kernel memory. For the inactive partition, both contexts are stored in kernel memory. The hypercalls are used by the partitions to invoke the kernel by executing the software interrupt instruction. The kernel provides two types of hypercalls: *message sending hypercall* and *status switching hypercall.* To send a message, the partition executes the instruction "SWI 1". The software interrupt handler stores the message into the message-box of the receiver and restores the sender. The status switching hypercall changes the executing status of the partition by executing "SWI 0". The kernel backs up the CPU state into its own memory and reactivates the interrupted partition.

The irq-handler implements a static round-robin scheduler, that suspends the active partition and resumes the other one. It is also in charge of delivering the pending messages to the resuming (receiver) partition; if the message-box of the resuming partition is full, (i) its status is changed to "message", (ii) the context of its message status is updated with the content of the pending message, and (iii) the program counter of the resumed partition is updated to point to its message handler code. The reception of a message causes the resumed partition to enter into a local critical section, i.e. no other message can be received while the partition is running in the message status. To exist from the critical section, the receiver partition performs a status switch hypercall.

To start the system, a memory image of the system must be prepared by the linker. Let $mem_1 : [min_1, max_1] \rightarrow$ *word8* and $mem_2 : [min_2, max_2] \rightarrow$ *word8* be two initial partition memories in our setting. Then, the linker loads the initial partition memories and the kernel memory into the system memory, and activates the kernel bootstrap code. The *initial state* of the real system is the first reachable state in user mode obtained after the execution of the bootstrap code of the kernel. Clearly, this initial state of the system depends on the partitions memories. We denote the behavior of a real system starting from an initial state s_0 with initial partition memories mem_1 and mem_2 by the transition system $T_r(mem_1, mem_2) = \langle \mathcal{S}, s_0, \rightarrow \rangle$.

4. THE IDEAL SYSTEM

The *ideal system* formalizes the top level specification which satisfies the required separation properties by construction. The ideal system is composed of two separate special ARMv7 machines communicating via asynchronous message passing, a logical component and a shared timer (see Fig. 1). Each machine of the ideal system is used to execute one of the two partitions in a physically isolated environment. Intuitively, the logical component can be considered as an external device. Our special ARMv7 machine allows a partition to execute without the runtime support of the kernel. This machine executes the user-mode computations as a regular ARMv7 processor, but if the processor switches to a privileged mode, an abstract kernel functionality is atomically executed and the user mode is restored.

An ideal state is a record $q = \langle \sigma_1, \sigma_2, c_1, c_2, t, id \rangle \in \mathcal{Q}$ where t represents the clock cycles elapsed from the system start. At each instant, only one of the machines can perform computations, and $id \in \{1, 2\}$ identifies the active machine. The logical component consists of the records $c_i = \langle rdy, ctx, msg \rangle$, $i \in \{1, 2\}$. The flag *rdy* represents if the machine is ready to handle the incoming messages and the banked context *ctx* (set of registers and cpsr) is used to back up the machine state whenever a message is received. A message box $msg \in word32 \cup \{\perp\}$ can either contain a pending message or be empty ($\perp$). Henceforth, we say that an ideal system is in mode m if the active machine is in mode m and the inactive one is in the mode *usr*.

The initial state of the ideal system, similar to the real system, depends on the initial partition memories. We denote the behavior of a ideal system starting from an initial state q_0 with initial partition memories mem_1 and mem_2 by the transition system $T_i(mem_1, mem_2) = \langle \mathcal{Q}, q_0, \rightarrow \rangle$. In the initial state, all the components are initialized such that the partitions memories and the page tables are loaded into their corresponding machine memory, the program counter

$$\frac{M(\sigma_1) = \mathit{usr} \wedge \langle \sigma_1, t\rangle \rightarrow \langle \sigma_1', t'\rangle}{\langle \sigma_1, \sigma_2, c_1, c_2, t, 1\rangle \rightarrow \langle \sigma_1', \sigma_2, c_1, c_2, t', 1\rangle}\ \mathit{UserR} \qquad \frac{M(\sigma_1) = \mathit{irq} \wedge (c_2.\mathit{msg} = \perp \vee \neg c_2.\mathit{rdy})}{\langle \sigma_1, \sigma_2, c_1, c_2, t, 1\rangle \rightarrow \langle \mathit{RestoreUser}(\sigma_1), \sigma_2, c_1, c_2, t + t_{sh}, 2\rangle}\ \mathit{SchR}$$

$$\frac{\begin{array}{c} M(\sigma_1) = \mathit{svc} \wedge \mathit{curr}(\sigma_1) = \mathit{SWI\ 0} \\ (\sigma_1', c_1') = \mathit{Switch}(\mathit{RestoreUsr}(\sigma_1), c_1) \end{array}}{\langle \sigma_1, \sigma_2, c_1, c_2, t, 1\rangle \rightarrow \langle \sigma_1', \sigma_2, c_1', c_2, t + t_{switch}, 1\rangle}\ \mathit{SwitchR} \qquad \frac{\begin{array}{c} M(\sigma_1) = \mathit{irq} \wedge c_2.\mathit{rdy} \wedge c_2.\mathit{msg} \neq \perp \\ (\sigma_2', c_2') = \mathit{Receive}(\sigma_2, c_2) \end{array}}{\langle \sigma_1, \sigma_2, c_1, c_2, t, 1\rangle \rightarrow \langle \mathit{RestoreUsr}(\sigma_1), \sigma_2', c_1, c_2', t + t_{rcv}, 2\rangle}\ \mathit{RcvR}$$

$$\frac{\begin{array}{c} M(\sigma_1) = \mathit{svc} \wedge \mathit{curr}(\sigma_1) = \mathit{SWI\ 1} \\ c_2' = \langle c_2.\mathit{rdy}, c_2.\mathit{ctx}, \mathit{Out}(\sigma_1)\rangle \end{array}}{\langle \sigma_1, \sigma_2, c_1, c_2, t, 1\rangle \rightarrow \langle \mathit{IncPC}(\mathit{RestoreUsr}(\sigma_1)), \sigma_2, c_1, c_2', t + t_{snd}, 1\rangle}\ \mathit{SendR}$$

Figure 3: Semantics of the Ideal System

of each machine points to the entry point of the corresponding partition, and both machines are in the mode *usr*.

Figure 3 shows the semantics of the ideal system when machine 1 is active. Due to lack of space, we only present the semantics of hypercalls and scheduling. In all cases, the ideal transitions yield the two machines in user mode. The rules for machine 2 active are similar. Each kernel functionality f comes with a fixed time budget t_f for its execution.

The rule *UserR* states that if the processor is in user mode, execution of an instruction does not affect the state of the logical component and the inactive machine, and it behaves as if it is executed on a regular ARMv7 machine. Instructions consume the same amount of cycles in the real and ideal systems.

A machine is rescheduled whenever a hardware interrupt is triggered by the timer. The function *RestoreUser* restores user mode and the corresponding banked registers. The rule *SchR* describes that if there is no message for the resumed machine or the resumed machine is not ready to handle a message, the active partition is changed to the current suspended one. The rule *RcvR* expresses the reception of a pending message by the resumed machine in which the function $\mathit{receive}(\sigma_2, c_2)$ disables the *rdy* flag for entering into a critical section, moves the pending message into the input buffer of the receiver, and sets the program counter to point to the message handler of the receiver.

The rule *SendR* describes the semantics of message sending (activated due to the execution of "SWI 1" in the previous state) that copies the message into the message box of the inactive machine. If the status switch hypercall is invoked, the rule *SwitchR* toggles the *rdy* flag and restores the banked context.

Figure 2 (B) depicts an example computation of an ideal system. In the states 2, 6 and 12 the system traps an exception raised on the active machine and atomically applies an ideal kernel functionality.

5. PROOF STRATEGY

To prove that the real model does not introduce information channels not already present in the ideal model it suffices to show that the observable traces for each partition are the same in both cases. In order to pin down this concept we need to define when each partition system is in control of the system, and what its observations are.

Top Level Proof Goal.

Intuitively, for the real system, partition g is in control when its program counter points to a location in mem_g. However, this is not entirely accurate. Instead we say that partition g is *active* in state s, $\mathit{act}_g(s)$, if the processor is in user mode and the status of the domain g, held by coprocessor register c_3 is client. For the ideal system, partition g is *active*, $\mathit{act}_g(q)$, if the *id* field of the ideal state is g and the corresponding machine is in the user mode.

The observations of partition g in real state s, assuming that g is active, are the CPU and memory resources observable by g in state s. This is the structure $O_g(s) = \langle \mathit{uregs}, \mathit{cpsr}, \mathit{mem}_g\rangle$ of user registers, cpsr and partition memory in state s. If g is inactive in s, g's observations are the user registers and cpsr of the saved context of g along with its partition memory. For the ideal system, $O_g(q)$ is the user registers, cpsr and the memory allocated to g of the corresponding machine in q.

Consider now an infinite execution $\pi_r = s_0 \rightarrow s_1 \rightarrow \cdots$ of the real system. The g-trace of π_r is the sequence $\omega(\pi_r, g)$ of observations obtained by first projecting out those states for which g is not active, and secondly extracting g's observations, or in other words, $\omega(\pi_r, g) = \mathit{map}(O_g, \mathit{prj}(\pi_r, \mathit{act}_g))$ where *prj* and *map* are the obvious projection/filtering functions. Similarly, if π_i is an ideal execution, the g-trace of π_i is $\omega(\pi_i, g) = \mathit{map}(O_g, \mathit{prj}(\pi_i, \mathit{act}_g))$.

Let now $\mathit{tr}_{g,r}(\mathit{mem}_1, \mathit{mem}_2)$ and $\mathit{tr}_{g,i}(\mathit{mem}_1, \mathit{mem}_2)$ be the set of g-traces of the real system and the ideal system with the initial partition memories of mem_1 and mem_2, respectively, and for arbitrary states s, q, let $\mathit{tr}_g(s)$ and $\mathit{tr}_g(q)$ be the sets of g-traces of executions starting in s, q, respectively. The top level proof goal is thus to prove that the sets of g-traces of the real and the ideal systems are identical, for $g \in \{1, 2\}$ and any arbitrary mem_1 and mem_2, or, more precisely, that

$$\mathit{tr}_{g,r}(\mathit{mem}_1, \mathit{mem}_2) = \mathit{tr}_{g,i}(\mathit{mem}_1, \mathit{mem}_2) \qquad (1)$$

for all initial partition memories mem_1, mem_2. If (1) holds we say that the PROSPER kernel guarantees isolation.

In order to prove (1), we first present three general lemmas concerning the safe executions of an ARM machine in user mode and its safe mode switching from user mode to privileged mode. Given the general ARM Lemmas, we prove lemmas for the real and the ideal systems to ensure correct initialization, correct userland execution, and isolation-guaranteed execution of the kernel handlers. We then proceed to present the proof of (1).

ARM Lemmas.

Our proof strategy identifies three lemmas concerning the ARM instruction set architecture that may have significance beyond the verification exercise reported here. The first is a general noninterference lemma stating that if an ARM machine executes in user mode in a memory protected configuration as studied here, the behavior of the active partition is influenced only by those resources allowed to do so. The predicate $sim_g(s_1, s_2)$ indicates that the status of the domain g held by coprocessor register c_3 is client in the user mode states s_1 and s_2, and they have the same user registers, cpsr, MMU configurations and the memory allocated to the domain g.

LEMMA 1. *If $sim_g(s_1, s_2)$ and $s_1 \rightarrow s'_1$, there exists s'_2 s.t. $s_2 \rightarrow s'_2$ and $sim_g(s'_1, s'_2)$, and vice versa.*

The second lemma establishes $\mathit{unmodified}_g(s, s')$ stating that the non-accessible resources for a state s in user mode, including the privileged psrs/registers, coprocessor registers, interrupt flags and the memory regions not allocated to the active partition g, are not modified in a transition from s to another user mode state s'. If s' is in privileged mode m, the privileged registers, psr_m and the interrupt flags, are excluded from the non-modifiable resources. We obtain:

LEMMA 2. *If $s \rightarrow s'$ and $act_g(s)$ then $\mathit{unmodified}_g(s, s')$.*

The predicate $\mathit{priv\_const}_g(s, s')$ asserts that if an ARM machine switches from s in user mode to s' in privileged mode m then the conditions for the execution of the handler are prepared properly, e.g., the program counter points to the correct entry of the vector table, the link register of m contains the correct return address of the partition, and the flags of *cpsr* and psr_m are set correctly.

LEMMA 3. *Suppose $s \rightarrow s'$, $act_g(s)$ and $M(s') \neq usr$. Then $\mathit{priv\_const}_g(s, s')$.*

(g, m)-*Compatible States.*

We then turn to the conditions needed to ensure that the partition observations are the same in the real and the ideal systems. These conditions depend on many aspects of the machine states and we are only able to outline the conditions here.

These conditions are complex because several elements can directly or indirectly influence the behavior of a partition. We briefly define the conditions for the user mode states (i.e. when all machines are in the user mode) and the switched mode states (i.e. the inactive machine of ideal machine is in the user mode but the real system and the active machine of ideal system have recently switched to the privileged mode).

Say that two states s and q are (g, m)-*compatible* if (i) s and q are in the mode m, (ii) g is the last active partition in s and q, i.e. the status of the domain g, held by coprocessor register c_3 is client in s, and the *id* field of the ideal state is g, (iii) the partitions have the same observations in s and q, $O_{g'}(s) = O_{g'}(q)$ for $g' \in \{1, 2\}$, (iv) the values of data-structures in the logical component of state q agree with the values of corresponding data structures in the kernel of state s, e.g. the message box of a partition in the kernel and the logical component contain the same values, (v) the MMU and coprocessors are configured correctly in all three machines, (vi) a set of invariants are held by the kernel data-structure in s, (vii) the interrupt flags are set correctly, e.g. the fast interrupt flag F of all machines are disabled, (viii) if m is a privileged mode then psr_m and the link register of the mode m must be identical in the active machine of q and s, to make sure that g is restored properly, (ix) s and q have the same system clock.

User/Handler Lemmas.

The relation of (g,m)-compatibility is our candidate unwinding relation, i.e. it is in some suitable sense which we go on to make precise preserved under computation. This involves showing the following two key properties:

- User Lemma: Each user mode transition in the real system is matched (in the sense of (g,m)-compatibility) by a corresponding user mode transition in the ideal system without interfering with the resources that are not intended to be accessible by the partition, and vice versa.
- Handler Lemma: Each complete handler execution in the real system is matched (as (g,m)-compatibility) by a corresponding execution of a kernel functionality in the ideal model, and vice versa.

Similar to the $\mathit{unmodified}_g$ predicate for the real system above, $\mathit{unmodified}_g(q, q')$ holds if the logical component, the inactive machine, and the non-accessible resources of the active machine are unmodified by an ideal transition from q to q' when g is active in q. The User Lemma then follows from the first and second ARM Lemmas as follows:

LEMMA 4 (USER). *For all (g, usr)-compatible states s and q, if $q \rightarrow q'$ then there exist s' and m such that*

(i) $s \rightarrow s'$,

(ii) the states s' and q' are (g, m)-compatible,

(iii) $\mathit{unmodified}_g(s, s')$, and (iv) $\mathit{unmodified}_g(q, q')$.

Vice versa, if $s \rightarrow s'$ then q' and m exists such that $q \rightarrow q'$ and the above properties (ii) and (iii) hold.

For the Handler Lemma we need to ensure that execution of the kernel handlers terminates, and that the compatibility conditions are satisfied when the control returns back to the partitions. Let $s_0 \rightsquigarrow s_n$ if there is a finite execution $s_0 \rightarrow \cdots \rightarrow s_n$ such that $M(s_n) = usr$ and $M(s_j) \neq usr$ for $0 < j < n$. Similarly, we define $\rightsquigarrow$ for the ideal system. These state relations are represented in Fig. 2 by the dashed arrows. The additional black states in the real world are internal kernel steps that can not be observed by the partitions.

LEMMA 5 (HANDLER). *Suppose that s and q are two (g, m)-compatible states, $m \neq usr$. Assume s and q are respectively reached by a transition from the states s' and q', and $\mathit{priv\_const}_g(q, q')$ and $\mathit{priv\_const}_g(s, s')$ hold. If $q' \rightsquigarrow q''$ then there exist s'' and $g' \in \{1, 2\}$ such that $s' \rightsquigarrow s''$, and the states s'' and q'' are (g', usr)-compatible, and vice versa.*

Furthermore, it is to be guaranteed that the initial states of the real and the ideal systems are compatible. That is, we must verify that the MMU is set up according to our model requirements, the kernel invariants are satisfied, the partitions memory and the interrupt flags are configured correctly. In addition, we must make sure that the kernel code is loaded in the right part of the memory.

PROPOSITION 1. *For all initial partition memories mem_1, mem_2, the kernel boot terminates in the state s_0, and there exists $g \in \{1,2\}$ s.t. s_0 and q_0 are (g, usr)-compatible*

Proof of Main Theorem.

We can now proceed to prove (1). This is almost done once we show that the initial states are related by a bisimulation relation of a suitable form. To this end say that a relation R on pairs (s,q) of user mode states is a *candidate relation*, if whenever sRq then for some $g \in \{1,2\}$, (i) $act_g(s)$, (ii) $act_g(q)$, (iii) $O_g(s) = O_g(q)$, and (iv) if $q \rightsquigarrow q'$, then there exists s' such that $s \rightsquigarrow s'$ and $s'Rq'$, and (v) vice versa, if $s \rightsquigarrow s'$, then there exists q' such that $q \rightsquigarrow q'$ and $s'Rq'$.

Note that for the user mode states s and s', if $s \to s'$ then $s \rightsquigarrow s'$. In Fig. 2, dotted lines exemplify a bisimulation relation. It is easy to check that the existence of a candidate relation is sufficient to ensure (1). In particular:

PROPOSITION 2. *Suppose that sRq for some candidate relation R. Then $tr_g(s) = tr_g(q)$.*

THEOREM 1. *The PROSPER separation kernel guarantees isolation.*

The proof of 1 obviously relies on the proofs of the above lemmas, which in turn, for Handler Lemma and the Lemma 1, relies on verification of the handler and the bootstrap code, as outlined in Section 7. But, it may be illustrative to explain how these lemma come together to allow the main Theorem 1 to be proved.

By Theorem 1 it suffices to find a candidate relation relating the initial states s_0 and q_0. Define the candidate relation as follows:

$$R = \{(s,q) | (g, usr)\text{-}compatible(s,q) \land g \in \{1,2\}\}$$

We get s_0Rq_0 by prop. 1.

To prove that R is a candidate relation, assume that sRq. Then s and q are (g, usr)-*compatible*. Thus, $act_g(s)$, $act_g(q)$ and $O_g(s) = O_g(q)$ hold.

Suppose now that $q \rightsquigarrow q'$. There are two cases:

Case 1: If the ideal transition does not involve mode switching it follows from the User Lemma that s' exists such that $s \rightsquigarrow s'$ and q' and s' are (g, usr)-*compatible*, whence $s'Rq'$ as desired.

Case 2: If the ideal transition involves a switch to privileged mode m, it follows from the User Lemma that the real and ideal system evolve to the (g,m)-*compatible* states s'' and q''. According to the Third ARM Lemma, these transitions are performed safely, i.e. $priv\_const_g(s, s'')$ and $priv\_const_g(q, q'')$ hold. From the Handler Lemma, we can conclude that there exist (g', usr)-*compatible* states s' and q' where $s'' \rightsquigarrow s'$, $q'' \rightsquigarrow q'$ and $g' \in \{1,2\}$. But then $s'Rq'$, as desired.

The converse direction, that $s \rightsquigarrow s'$ implies $q \rightsquigarrow q'$ and $s'Rq'$ for some q follows by a symmetric argument (or, in this simple case, by determinacy of the $\rightsquigarrow$ relation). This concludes the proof of Theorem 1.

6. ISOLATION PROPERTIES

The main theorem shows that the real system does not leak more information than the ideal system, under the caveats we have imposed. However, it may not be clear what information is leaked by the ideal system itself. Neither may it be clear how leakage properties of the ideal system can be transferred to the real system. In this section we throw light on these two issues.

Data Separation.

Concerning the ideal system itself we use the approach of [12] to analyze kernel data separation properties. Let $\mathcal{Q}_c = \{q | \exists s.\ sRq\}$ be the image of the candidate relation.

The *No-Exfiltration* property guarantees that a transition with the partition g active in its target, does not modify the resources of the other partition, except its communication channel, i.e. the message box:

LEMMA 6. *Let $g, g' \in \{1,2\}$, $g' \neq g$ and $q \in \mathcal{Q}_c$. If $q \rightsquigarrow q'$ and $q'.id = g$, then $O_{g'}(q) = O_{g'}(q')$, $q.c_{g'}.rdy = q'.c_{g'}.rdy$ and $q.c_{g'}.ctx = q'.c_{g'}.ctx$.*

The *No-Infiltration* property is a noninterference property guaranteeing that a transition for which g is active in its target state, depends only on the partition observations, its logical component and the MMU configuration. In particular, a transition ending in a state with the partition g active, is not influenced by data owned by the other partition.

LEMMA 7. *Let $q_1, q_2 \in \mathcal{Q}_c$ such that $q_1.c_g = q_2.c_g$, $q_1.t = q_2.t$, $q_1.id = q_2.id$ and $O_g(q_1) = O_g(q_2)$. If $q_1 \rightsquigarrow q_1'$, $q_2 \rightsquigarrow q_2'$, $act_g(q_1')$ and $act_g(q_2')$ then $q_1'.c_g = q_2'.c_g$, $q_1'.t = q_2'.t$ and $O_g(q_1') = O_g(q_2')$*

Similar properties can be proven for the real system using the candidate relation and the properties of the ideal system. Let $\mathcal{S}_c = \{s | \exists q.\ sRq\}$ be the preimage of the candidate relation, and the function $lc_g(s)$ extracts from the kernel memory the content of the data-structure that corresponds to c_g in the logical component. The following corollary states the no-infiltration and no-exfiltration for the real system.

COROLLARY 1. *Let $s_1, s_2 \in \mathcal{S}_c$, $s_1.t = s_2.t$, $act_g(s_1) \Leftrightarrow act_g(s_2)$.*

- *Suppose if $g' \neq g$, $s_1 \rightsquigarrow s_1'$ and $act_g(s_1')$ then $O_{g'}(s_1) = O_{g'}(s_1')$, $lc_{g'}(s_1).rdy = lc_{g'}(s_1').rdy$ and $lc_{g'}(s_1).ctx = lc_{g'}(s_1').ctx$.*
- *Suppose if $lc_g(s_1) = lc_g(s_2)$, $O_g(s_1) = O_g(s_2)$, $s_1 \rightsquigarrow s_1'$ and $s_2 \rightsquigarrow s_2'$, $act_g(s_1')$ and $act_g(s_2')$ then $lc_g(s_1') = lc_g(s_2')$ and $O_g(s_1') = O_g(s_2')$*

We sketch the proof of the second statement. Since s_1 and s_2 are in $\mathcal{S}_c$, then there exist q_1 and q_2 s.t. s_1Rq_1 and s_2Rq_2. We follow from the assumptions and the definition of R that $q_1.c_g = q_2.c_g$, $q_1.t = q_2.t$, $O_g(q_1) = O_g(q_2)$ and $q_1.idx = q_2.idx$. Since the candidate relation is a bisimulation, then for $j = 1,2$, there exists q_j' s.t. $q_j \rightsquigarrow q_j'$ and $s_j'Rq_j'$. Thus, $act_g(q_j')$, $lc_g(s_j') = q_j'.c_g$ and $O_g(s_j') = O_g(q_j')$. We conclude the proof by showing that $O_g(q_1') = O_g(q_2')$ and $q_1'.c_g = q_2'.c_g$ according to the Lemma 7.

Noninterference.

The no-exfiltration/no-infiltration properties give limited data separation properties at the level of single transitions. They do not, however, lift to executions, because messages may be passed between partitions which can introduce explicit data dependencies. As a sanity check, we therefore show how the security condition is reduced to noninterference in the special case where no exception except timer signal takes place. This property formalizes the intuition that if the partition g does not communicate, then the execution of the other partition is completely independent of activities of g.

A partition with the initial memory *mem* is called non-communicating, if for all arbitrary mem' and all states q that are reachable from the initial state of $T_i(mem, mem')$, $M(q.\sigma_1) = \{usr, irq\}$ holds.

THEOREM 2. *For any two non-communicating partitions with the initial memories* mem_1 *and* mem'_1, *and an arbitrary partition with the initial memory* mem_2,

$$tr_{1,i}(mem_1, mem_2) = tr_{1,i}(mem'_1, mem_2)$$

The symmetric theorem is proved when the non- communicating partition is deployed on the second machine. The state of a machine can be externally changed only by the reception of a message. Since the non-communicating partition never raises an exception, it can not execute the software interrupt and it can not send a message. Moreover, the system clock, shared between the two machines, must be independent of the activity performed by the non-communicating machine. This is possible because we assume that all transitions, with the exception of the ideal functionalities, require one clock cycle. Note that the candidate relation allows Theorem 2 to be directly transferred to the real system. The details are left out.

7. PROOF IMPLEMENTATION

The overall proof is carried out in the HOL4 theorem prover, following the proof strategy presented in Section 5. For those parts (the Handler Lemma and Proposition 1) that depend on kernel code we generate contracts and transfer the verification to BAP. To realize this we have produced a number of helper tools of which the main ones are: (i) an ARMv7 prover tool implemented in SML/HOL4, (ii) various tools and tool components interfacing HOL4 with BAP, (iii) a lifter tool to convert ARM assembly to BAP's input language. Space prevents us from more than outlining the BAP extensions and the proof of the ARM Lemmas here; these will be reported in separate publications.

7.1 Verification in HOL4

Overview of the ARM Model.

We use Fox et al's monadic HOL4 model of the ARMv7 instruction set architecture. The model has been validated against a development board, giving some credence to its accuracy [7].

A *computation* in the monadic HOL4 ARM model is a term of type

$$\alpha\ M = \texttt{arm\_state} \mapsto (\alpha \times \texttt{arm\_state})\texttt{error\_option}.$$

Computations act on a state `arm_state` and return either `ValueState` $a\ s$, a new state s of type `arm_state` along with a return value a of type α, or an error `Error` e. Errors represent all unpredictable computations, i.e., those that are underspecified by the ARM specification. The monad unit injects a value into a computation, while binding is a sequential composition operation which passes the return value of the first computation to the input parameters of the second one as follows:

```
f ≫= g = (λs.  case f s of Error e ↦ Error e
                || ValueState y t ↦ g y t )
```

The execution of an ARM instruction is defined by the computation `arm_next` modeling the entire processing of an instruction, from fetching the instruction pointed to by the program counter to the actual instruction execution.

The MMU Extension.

We extend the ARM model to support the MMU functionality in our setting. Given the complexity of memory management, the model is restricted to support only those parts of the MMU functionality used by the PROSPER kernel.[2] We also proved that the MMU configurations of all reachable states are supported by the extended model and not underspecified. The original ARM model tracks the history of memory accesses, allowing to compute the set of memory pages accessed by an instruction. To be accurate, it is necessary to check the access list after each primitive computation. To this end, the monadic structure is modified so that access history checks are introduced at every sequential composition of two computations. In case of an access violation within the first computation, the second one is simply disregarded, returning the unspecified value *ARB* along with the first state where an access violation has been recorded.

```
f ≫= g = (λs.  case f s of Error e ↦ Error e
               || ValueState y t ↦
               (if (access_violation t)
               then (ValueState ARB t)
               else (g y t)) )
```

Proof of the ARM Lemmas.

We use a relational Hoare logic framework to prove the ARM Lemmas. For technical reasons we formulate the three ARM Lemmas as a single statement. For any computation $\mathtt{f}$ and predicates $\mathtt{p}_1$, $\mathtt{p}_2$, we define a relational predicate $\mathtt{preserving}(\mathtt{f}, \mathtt{p}_1, \mathtt{p}_2)$ stating that, when starting from two states in the relation sim_g and satisfying $\mathtt{p}_1$, then the states returned by $\mathtt{f}$ (i) are in the relation sim_g, (ii) satisfy the non-modification and mode-switching constraints, as presented in Section 5, and (iii) satisfy $\mathtt{p}_2$. The state predicates $\mathtt{p}_1$ and $\mathtt{p}_2$ allow processor mode specific reasoning. The final goal is to show that the MMU-enabled variant of `arm_next` satisfies `preserving` when starting from user mode.

A set of sound inference rules have been implemented in a semi-automatic HOL4 helper tool. An example is the rule for sequential composition:

$$\frac{\mathtt{preserving}(\mathtt{f}_1, \mathtt{p}_1, \mathtt{p}_1) \qquad \mathtt{preserving}(\mathtt{f}_2, \mathtt{p}_1, \mathtt{p}_2)}{\mathtt{preserving}(\mathtt{f}_1 \gg= (\lambda \mathtt{x}.\mathtt{f}_2), \mathtt{p}_1, (\mathtt{p}_1 \vee \mathtt{p}_2))}$$

[2] Only section-based one-level page tables without address translation are supported so far.

The tool recognizes the structure of a computation, decomposes the verification goal in a set of sub-goals, proves the sub-goals recursively and applies the suitable inference rule to infer the initial goal. Moreover, it searches in the HOL4 database and the user-provided theorems to find a suitable theorem that can prove the goal. We prove `preserving` for the write primitive computations manually, but the tool can handle some read computations automatically, allowing to prove a large share of the workload automatically.

Generation of Pre- and Postconditions.

HOL4 is also used to generate pre- and postconditions for the kernel handlers, for subsequent verification with BAP. Consider a handler with the starting state s_1 in mode m such that $s_1 \rightsquigarrow s_2$. Suppose that s_1 and q_1 are (g, m)-compatible such that q_1 is the starting state of the corresponding ideal handler functionality. Let q_2 be a state such that $q_1 \rightsquigarrow q_2$. These conditions allow to automatically generate the precondition of the handler under which the final state s_2 will be (g, usr)-compatible with q_2. The preconditions are generated by the hypotheses of the Handler Lemma: the starting state of the kernel handler s_1 is (g, m)-*compatible* with q_1, and there are s_0 and q_0, such that $priv\_const_g(q_0, q_1)$, $priv\_const_g(s_0, s_1)$, and s_0 and q_0 are in the candidate relation.

7.2 Binary Code Verification

The kernel code verification relies on Hoare logic. To prove the Handler Lemma and Proposition 1, we are required to verify several Hoare triples $\{P\}C\{Q\}$ for the exception handlers and the bootstrap code, that is we check that if the precondition P holds in the starting state of C, then the postcondition Q is guaranteed by C. When possible, we adopt a standard semi-automatic strategy, i.e. firstly, we compute the weakest liberal precondition $wlp(C, Q)$ on the starting state, then prove that the precondition P implies the weakest precondition. This task can be fully automated if the predicate $P \implies wlp(C, Q)$ is equivalent to a predicate of the form $\forall x.A$ where A is quantifier free. The validity of A can then be checked using a Satisfiability Modulo Theory (SMT) solver that supports bitvectors to handle operations on words. In this work, we used STP [9].

Weakest preconditions can be computed directly in HOL4 using the ARMv7 model. However, this task requires a significant engineering effort. We adopted a more practical approach, by using (BAP) [6]. The BAP toolset provides platform-independent utilities to extract control flow graphs and program dependence graphs, to perform symbolic execution and to perform wp calculations. These utilities reason on the BAP Intermediate Language (BIL), a small and formally specified language that models instruction evaluation as compositions of variable reads and writes in a functional style.

We found the existing BAP front-end to translate ARM programs to BIL inadequate for our purpose: It supports only ARMv4, it does not manage the processor status registers, and it does not handle banked registers for the privileged modes and coprocessor management. To this end, we developed a new front-end for ARMv7 programs using the ARM model available in HOL4. This tool allows us to translate the code of the kernel handlers and the bootstrap into BIL.

The HOL4 ARM model provides the function `arm_steps` to compute the set of pairs $\langle c_1, t_1\rangle, \dots \langle c_n, t_n\rangle$ for an instruction where the function t_i transforms a state provided that the condition c_i holds on that state. In other words, $\forall$`s:arm_state arm_next s = ValueState ()` t_i(`s`) if c_i(`s`). In order to use `arm_steps`, the execution mode and the instruction set type (e.g. Thumb, ARM) must be known. Our handlers preconditions set the value of these parameters. The translation from ARM to BIL is performed by translating the HOL4 conditions c_i and functions t_i to BIL fragments.

Verifying the Hoare triples using weakest preconditions requires us to handle some common issues. Algorithms to compute weakest preconditions rely on the absence of indirect jumps, i.e. the jumps whose target address is a variable. We used the SMT solver to automatically compute jump targets, depending on the instruction precondition. Weakest preconditions can grow exponentially in the number of instructions. We extended BAP to simplify the weakest precondition during its backward propagation using ARM specific simplification patterns. Finally, our verification task has been simplified by the structure of the kernel code. All loops of the kernel have a single control flow node that represents both the entry point and the exit point of the loop. In the general case, we defined a loop invariant and a loop variant and applied the standard Hoare logic rules to prove the contract. Verification has been simplified by the absence of loops in the kernel handlers and the fact that the boot code contains only for-loops that iterate over integer sets.

8. EVALUATION

We tested the kernel implementation using OVP [21] as main execution environment, which provides a simulation infrastructure convenient to evaluate our kernel. Slightly different versions of the kernel have been deployed on Beagleboard, Beagleboard-XM, Beaglebone, NovaThor and the Integrator development board. The size of kernel code, internal data-structures and page table are respectively less than 4 kB, 2 kB and 16 kB. The main functionality of the kernel is provided by the software and hardware interrupt handlers. In the worst case, the hardware interrupt handler executes 112 instructions, including 48 reading and 22 writing accesses to the memory. Similarly, the software interrupt handler executes at most 46 instructions, including 20 reading and 8 writing accesses to the memory. All the memory locations accessed by the kernel handlers belong to the internal kernel data-structures. To minimize the system overhead and avoid accesses to system memory during kernel tasks, we can use scratchpad memory or cache locking due to the size of the run-time footprint.

We identified and fixed several bugs in the kernel implementation during the verification process: (i) the registers were not sanitized after the bootstrap, (ii) some of the execution flags were not correctly restored during the context switch, (iii) the procedure to decode the hypercall identifier did not consider the case that the partition is running in thumb execution mode.

The model of the ideal system, the formalization of the verification procedure and the proofs of the theorems consist of 21k lines of HOL4 code. The tools developed to support the verification of the kernel contracts required 2k lines of HOL4 and python code. The kernel binary code is verified with respect to sixteen contracts, each of them consisting of $\sim$ 400 lines of assertions that are automatically generated from HOL4. In the worst case, the verification of one contract required $\sim$ 30 minutes using one Intel(R) Xeon(R) X3470 core; the contract is generated in $\sim$ 5 minutes, the indirect jumps are solved in $\sim$ 2 minutes, the weakest precondition is computed in $\sim$ 10 minutes and the SMT solver verifies the validity of the resulting condition in $\sim$ 15 minutes.

9. RELATED WORKS

Past work on formal verification of kernel information flow properties [12, 19, 23, 4] are based on variants of noninterference [11]. Typically, the goal is to allow a number of component systems, partitions, or guest systems, depending on terminology, to share a computing platform without any interaction, leaving possible communication between the partitions to be managed by mechanisms outside the model. In Heitmeyer et al [12], for instance, partitions have explicit input and output buffers, but communication is delegated to external agents, in this way allowing properties like absence of infiltration (roughly: direct flows) and exfiltration (indirect flows) to be proved. Similar results are reported in [23, 4] and in [30] at the firmware level. Murray et al [20] considers noninterference in presence of a dynamic scheduler and uses a version of intransitive noninterference [24] (actually, NI) to allow a scheduler to influence which partition is scheduled, without permitting the scheduler to be used as a covert channel, as discussed briefly in the introduction.

Several recent works address hypervisor/microkernel verification, although without taking information flow into account. In [14] a simulation property of an entire microkernel down to a C implementation was verified using the Isabelle theorem prover. This work was recently extended to ARM assembly using decompilation techniques [29]. Alkassar et al [2] proposed an automatic approach to verify a hypervisor for a (simplified) MIPS machine by annotating the C code with contracts and checking them using VCC. They establish a reachability property: at any time the state of a partition can be reached by executing the same partition on a completely isolated machine. This is sufficient to establish simulation when the specification is deterministic (but not otherwise). To allow VCC to reason about statically unknown partitions/guests, a C emulator of the MIPS machine has been implemented and annotated. The C emulator has been adopted also to verify parts of the hypervisor that mix C and assembly code [22].

Most kernel security analyses address the kernel routines one at a time, using suitable relational specifications. Without verifying the correct interaction between the kernel routines and the processor (e.g. mode switching and memory protection), these specifications are not sufficient to guarantee security at system level, i.e. at the level of "full" executions that interleave kernel routines with userland execution of the partitions. Performing such a systems-level, integrated analysis (kernel and processor) has not been done before for realistic processor architectures. For instance [30, 19] address kernel routines but not the processor interaction. Analysing each kernel routine in isolation can be done using existing versions of conditional non-interference (as discussed in [19]). However, this does not guarantee information flow security at system level. Our approach to formulating the TLS using idealized userland processors solves this problem, simply by showing that the full executions for the ideal and the real model are the same.

Barthe et al. [4] formalized a hypervisor model using the Coq proof assistant. They focus on establishing that the hypervisor ensures isolation properties between the guests, abstracting away from actual hypervisor implementation.

10. DISCUSSION

We have presented a separation kernel, the PROSPER kernel, for ARMv7-A and a machine-assisted proof of information flow correctness using a combination of tools (HOL4 [13] and BAP [6]). Our analysis has a number of distinguishing features:

- Our top-level specification (TLS) and verification approach is deliberately designed to take inter-component communication into account, an ever present challenge in the verification of information flow properties for real systems.

- We introduce a new technique for building a TLS for this type of application, based on communicating idealized userland processors.

- The security analysis is performed at systems level, modeling both the MMU-constrained user space execution of arbitrary partitions, the kernel handlers, and the interaction of the two.

- We validate the "monotonicity or release" property, as suggested by Sabelfeld and Sands [26], by showing that the security proof reduces to standard noninterference for the special case of non-communicating partitions.

- The entire analysis is performed at machine code level for a commodity processor architecture.

A number of subsidiary contributions include several tools for managing and executing the proofs, including the ARM prover tool for verifying critical partition correctness properties of the ARMv7 machine architecture based on an extension of Fox and Myreen's monadic ARM semantics [7], and several extensions to the BAP toolset.

By verifying the entire kernel at machine code level we avoid reliance on a C compiler, and we can transparently verify code that mix C and assembly. Generally speaking, verification at machine code level is time consuming, however we were supported by the fact that the code was mostly compiler produced and loops were used in only a few places.

Since our TLS specifies the exact set of traces allowed by an implementation, a worry might be that the TLS becomes overly detailed and unwieldy. We did not find this to be the case. Rather, the development of the ideal model, as we progressed to understand the various issues involved, was a great help in organizing the thinking. It is true that our approach (as in other work, cf. [12]) precludes an abstract treatment of scheduling, but this is to be expected when information flow is to be taken into account.

On two counts our model is not yet satisfactory. The first concerns timing. Our model counts instruction cycles [3], instead of real clock cycles. In our implementation the former is used. It is non-trivial to extend our analysis to a more

[3] This is the element of our "real system" that is not really real.

realistic time representation, as in this case well-known phenomena such as cache delays and instruction pipelining come into play. Cache leakage has been considered in the context of virtualization by Barthe et al [5]. Zhang et al [31] demonstrated an access-driven side-channel attack targeting the Xen hypervisor. The authors (i) use interprocess interrupts to affect the Xen scheduler and to reduce the time slot available to the victim and (ii) indirectly monitor the usage of the instruction cache, which is shared among partitions. Extending our approach to handle access-driven attacks requires a more refined analysis of timing behaviour, which is part of our ongoing research efforts.

The second count is unpredictable states. According to the ARM Architecture Reference Manual [3], unpredictable behaviour is not allowed to "perform any function that cannot be performed at the current or lower level of privilege using instructions that are not unpredictable". This definition is difficult to accommodate in our framework. An interpretation of allowed behaviour which is adequate for our purpose is "compliant with the ARM Lemmas". This enables our proofs to go through, and in fact we posit that this may be a more helpful and less prescriptive definition than that of [3]. Practically, the ARM Lemmas can be used to certify if a specific ARMv7-A implementation can be used to host our kernel. In the proof implementation we have used the error states introduced in the monadic ARM HOL4 model. This is not really satisfactory, though, as this allows partitions to exit the scope of our model at will, by entering an unpredictable state. We leave a better treatment of unpredictable behaviour, in addition to more realistic hardware models and kernel functionality, to future work.

Finally we emphasize that virtualization and/or separation kernels are not the only tools available for secure partitioning. The ARM-proprietary TrustZone solution [1] adds to the standard ARMv7 architecture a secure partition that can be used to split the CPU resources between an untrusted and a trusted OS. Our results show that the kernel can be protected using standard, less expensive hardware, and a smaller TCB. Moreover, extending the proposed verification strategy can be straightforwardly extended to manage a different number (>2) of partitions.

11. ACKNOWLEDGMENTS

Work supported by framework grant "IT 2010" from the Swedish Foundation for Strategic Research.

12. REFERENCES

[1] ARM TrustZone technology. `http://www.arm.com/products/processors/technologies/trustzone.php`.

[2] E. Alkassar, M. A. Hillebrand, W. J. Paul, and E. Petrova. Automated verification of a small hypervisor. In *Proc. VSTTE*, volume 6217 of *Lecture Notes in Computer Science*, pages 40–54. Springer, 2010.

[3] ARMv7-A architecture reference manual. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c`.

[4] G. Barthe, G. Betarte, J. D. Campo, and C. Luna. Formally verifying isolation and availability in an idealized model of virtualization. In *Proc. FM'11*, volume 6664 of *Lecture Notes in Computer Science*, pages 231–245. Springer, 2011.

[5] G. Barthe, G. Betarte, J. D. Campo, and C. Luna. Cache-leakage resilient os isolation in an idealized model of virtualization. In *Proc. CSF'12*, pages 186–197, Washington, DC, USA, 2012. IEEE Computer Society.

[6] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A binary analysis platform. In *Proc. CAV'11*, volume 6806 of *Lecture Notes in Computer Science*, pages 463–469. Springer, 2011.

[7] 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.

[8] FreeRTOS. `http://www.freertos.org/`.

[9] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Proc. CAV'07*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531. Springer, 2007.

[10] C. Gehrmann, H. Douglas, and D. Nilsson. Are there good reasons for protecting mobile phones with hypervisors? In *Consumer Communications and Networking Conference (CCNC), 2011 IEEE*, pages 906 –911, jan. 2011.

[11] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[12] 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.

[13] HOL4. `http://hol.sourceforge.net/`.

[14] 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.

[15] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pages 104–113, London, UK, 1996. Springer-Verlag.

[16] 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.

[17] J. McDermott, B. Montrose, M. Li, J. Kirby, and M. Kang. Separation virtual machine monitors. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 419–428, New York, NY, USA, 2012. ACM.

[18] μClinux. `http://www.uclinux.org/`.

[19] T. C. 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 *IEEE Symposium on Security and Privacy*, pages 415–429. IEEE Computer Society, 2013.

[20] T. C. Murray, D. Matichuk, M. Brassil, P. Gammie, and G. Klein. Noninterference for operating system kernels. In *CPP*, volume 7679 of *Lecture Notes in Computer Science*, pages 126–142. Springer, 2012.

[21] Open Virtual Platforms. `http://www.ovpworld.org/`.

[22] W. J. Paul, S. Schmaltz, and A. Shadrin. Completing the automated verification of a small hypervisor - assembler code verification. In *SEFM*, volume 7504 of *Lecture Notes in Computer Science*, pages 188–202. Springer, 2012.

[23] 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.

[24] J. Rushby. *Noninterference, transitivity, and channel-control security policies*. SRI International, Computer Science Laboratory, 1992.

[25] J. M. Rushby. Design and verification of secure systems. In *Proc. SOSP'81*, pages 12–21, 1981.

[26] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. Comput. Secur.*, 17(5):517–548, Oct. 2009.

[27] R. Sailer, E. Valdez, T. Jaeger, R. Perez, L. V. Doorn, J. L. Griffin, S. Berger, R. Sailer, E. Valdez, T. Jaeger, R. Perez, L. Doorn, J. Linwood, and G. S. Berger. sHype: Secure hypervisor approach to trusted virtualized systems. In *IBM Research Report RC23511*, 2005.

[28] D. Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2012.

[29] T. A. L. Sewell, M. O. Myreen, and G. Klein. Translation validation for a verified os kernel. In *PLDI*, pages 471–482, 2013.

[30] M. M. Wilding, D. A. Greve, R. J. Richards, and D. S. Hardin. Formal verification of partition management for the AAMP7G microprocessor. In *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pages 175–191. Springer US, 2010.

[31] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM side channels and their use to extract private keys. In *Proc. CCS'12*, pages 305–316. ACM, 2012.