Metadata of the chapter that will be visualized in SpringerLink

Book Title	Verified Software: Theories, Tools and Experiments	
Series Title		
Chapter Title	Separation Kernel Verification: The Xtratum Case Study	
Copyright Year	2014	
Copyright HolderName	Springer International Publishing Switzerland	
Author	Family Name	**Sanán**
	Particle	
	Given Name	**David**
	Prefix	
	Suffix	
	Division	
	Organization	Trinity College Dublin
	Address	Dublin, Ireland
	Email	david.sanan@scss.tcd.ie
Corresponding Author	Family Name	**Butterfield**
	Particle	
	Given Name	**Andrew**
	Prefix	
	Suffix	
	Division	
	Organization	Trinity College Dublin
	Address	Dublin, Ireland
	Email	andrew.butterfield@scss.tcd.ie
Author	Family Name	**Hinchey**
	Particle	
	Given Name	**Mike**
	Prefix	
	Suffix	
	Division	
	Organization	University of Limerick
	Address	Limerick, Ireland
	Email	mike.hinchey@lero.ie

Abstract

The separation kernel concept was developed as an architecture to simplify formal kernel security verification, and is the basis for many implementations of integrated modular avionics in the aerospace domain. This paper reports on a feasibility study conducted for the European Space Agency, to explore the resources required to formally verify the correctness of such a kernel, given a reference specification and a implementation of same. The study was part of an activity called Methods and Tools for On-Board Software Engineering (MTOBSE) which produced a natural language Reference Specification for a Time-Space Partitioning (TSP) kernel, describing partition functional properties such as health monitoring, inter-partition communication, partition control, resource access, and separation security properties, such as the security policy and authorisation control. An abstract security model, and the reference specification were both formalised using Isabelle/HOL. The C sources of the open-source XtratuM kernel were obtained, and

an Isabelle/HOL model of the code was semi-automatically produced. Refinement relations were written manually and some proofs were explored. We describe some of the details of what has been modelled and report on the current state of this work. We also make a comparison between our verification explorations, and the circumstances of NICTA's successful verification of the sel4 kernel.

Separation Kernel Verification: The Xtratum Case Study

David Sanán[1], Andrew Butterfield[1(✉)], and Mike Hinchey[2]

[1] Trinity College Dublin, Dublin, Ireland
{andrew.butterfield,david.sanan}@scss.tcd.ie
[2] University of Limerick, Limerick, Ireland
mike.hinchey@lero.ie

Abstract. The separation kernel concept was developed as an architecture to simplify formal kernel security verification, and is the basis for many implementations of integrated modular avionics in the aerospace domain. This paper reports on a feasibility study conducted for the European Space Agency, to explore the resources required to formally verify the correctness of such a kernel, given a reference specification and a implementation of same. The study was part of an activity called Methods and Tools for On-Board Software Engineering (MTOBSE) which produced a natural language Reference Specification for a Time-Space Partitioning (TSP) kernel, describing partition functional properties such as health monitoring, inter-partition communication, partition control, resource access, and separation security properties, such as the security policy and authorisation control. An abstract security model, and the reference specification were both formalised using Isabelle/HOL. The C sources of the open-source XtratuM kernel were obtained, and an Isabelle/HOL model of the code was semi-automatically produced. Refinement relations were written manually and some proofs were explored. We describe some of the details of what has been modelled and report on the current state of this work. We also make a comparison between our verification explorations, and the circumstances of NICTA's successful verification of the sel4 kernel.

1 Introduction

The separation kernel concept was introduced by Rushby [1] to aid in achieving high assurance in critical systems. A separation kernel creates a secure environment providing temporal and spatial partitioning of applications. In this environment each application can only access the set of resources that the kernel assigns, by being isolated into partitions where there is no flow of data beyond explicitly authorized channels.

In the last decade the use of separation kernels has increased, with architectures such as the Multiple Independent Levels of Security and Safety architecture

Funded by ESTEC CONTRACT No. 4000106016, and supported, in part, by Science Foundation Ireland grant 10/CE/I1855

D. Giannakopoulou and D. Kroening (Eds.): VSTTE 2014, LNCS 8471, pp. 1–17, 2014.
DOI: 10.1007/978-3-319-12154-3_9

(MILS) [2], appearing, as well as standards such as Common Criteria (CC) [3] and the associated Separation Kernel Protection Profile (SKPP) [4] for security requirements, or ARINC-653 [5] for functional requirements. In particular, in the space environment, the IMA for Space (IMA-SP) platform proposed by the European Space Research and Technology Centre (ESTEC) uses a separation kernel, which helps with fault containment.

Based on those standards and architectures, different implementations of the separation kernel have emerged, such as PikeOS [6], seL4 [7], XtratuM [8], or Air2 [9] among others. The SKPP assurance requirements only require, from the formal methods point of view, that a certified kernel design is semiformally verified and to provide a formal security policy model. However, for simple and small designs it is technically possible to go beyond those levels and perform a full verification of the implementation to ensure the absence of errors.

A separation kernel implementation is too complex for a direct approach to the verification of the implementation's code. Aiming to reduce the verification complexity, projects like L4.Verified [7], VerisoftXT [6], and the verification of the software for the so called Embedded Device (ED) [10] have successfully applied techniques such as refinement verification, where the properties are verified over an abstract model, which is less detailed and therefore easier to verify. Then, by means of so-called forward simulation, the implementation model is verified. L4.Verified and VerisoftXT respectively aim to verify the general purpose kernels seL4 and PikeOS, each composed of around 10 K lines of code. On the other hand, the verification in [10] targets a software-based embedded device of around 3k lines of code. L4.Verified fully verified security and functional properties of the seL4 kernel down to the C implementation and machine code. VerisoftXT fully verifies functional correctness of PikeOS, including memory separation correctness as well. Finally the work in [10] is focused on Common Criteria certification and verification is restricted to security properties. The significance of secure micro-kernel verification is still increasing and new on-going projects continue to arise, like [11] where non-interference has been proved on Prosper, a simple separation micro-kernel targeting the ARM architecture, and [12] where the information flow policy for the SAFE security kernel was verified.

Within the project *Methods and Tools for On-Board Software Engineering (MTOBSE)*[1] we developed an abstract model which captures requirements of a reference specification, for a partition-separation kernel, which was also developed under this project, guided by the SKPP and the IMA-SP specification [13]. We also formalised an abstract model of a security policy, and we provide a formal refinement relation between this and the reference model. We selected the open source XtratuM kernel [8] as the code verification target. Using existing model extraction tools, we have partially modeled the XtratuM microkernel and we have formalised the refinement relation between the abstract model and the XtratuM implementation model.

Unlike L4.Verified, VerisoftXT, and the verification of the ED kernel, we are not either starting from a particular specification and deriving verifiable

[1] funded by ESTEC CONTRACT No. 4000106016

code (L4.Verified), or taking existing code and abstracting out a specification (VerisoftXT and ED kernel). In our case we have written a general reference specification and have constructed an abstract model of that specification. We are exploring the feasibility of verifying third-party kernel implementations, such as XtratuM, against our reference abstract model, which we expect to be more difficult as a result of the independent development of the model and implementation.

As work in progress, we describe the methodologies and toolchain we are using to formally verify the XtratuM kernel w.r.t. the abstract model, and we also detail the process used to obtain the implementation model from the source code, as well as the issues that arise regarding feasibility.

2 About the Partitioned-Separation Kernel: A Reference Specification

Before John Rushby introduced the concept of a separation kernel [1], security models for high assurance kernels were too complex for feasible sound verification. The separation kernel approach uses simpler notions of partition and separation to result in a security kernel for which proof is much easier, but the necessary characteristics are still covered. To achieve process isolation, a secure kernel considers the spatial separation of resources and temporal separation of execution. Spatial separation restricts the set of resources each process can access to a designated set, while temporal separation ensures the execution of each process occurs at well defined times, without being changed or delayed by activities elsewhere in the system. These two concepts ensure that processes have the perception of being executed in independent environments. However, it is necessary to allow flows of information among processes. Since those information flows break the spatial separation principle, the separation kernel protection profile (SKPP) [4] establishes a Partition Information Flow Policy (PIFP) that relaxes the separation, but only for appropriate information flows.

The natural language reference specification [14], developed in the MTOBSE project, consists of software requirements, interface requirements, and the architectural design. Both the software requirements and the interface requirements are based on the requirements baseline for the IMA-SP platform [13] and ESA's own suggestions. Based on the specified requirements, the architectural design describes the data structures, component internal interfaces, and component functionalities.

The main sources from which the software requirements are drawn are the Arinc-653 standard specification [5] for functional requirements (partitioning aspects) and the Separation Kernel Protection Profile (SKPP) [4] for security requirements (separation aspects). To achieve temporal partitioning as defined in the ARINC-653 standard, IMA-SP requires the use of a static cyclic scheduler, where partitions are assigned to execution windows called Partition Time Windows (PTW), which are strictly dispatched according to their assigned execution quotas and period. IMA-SP allows the static configuration of a set of schedulers,

and during execution system partitions can invoke system calls to change the active scheduler to one of that set. Additionally, spatial partitioning is achieved by memory protection hardware, and forbidding any partition communication not explicitly allowed by an inter-partition communication channel, enforced by the separation kernel's PIFP.

Although the developed reference specification is guided by these standards, it does not intend to be fully compliant with them. For example, a non-standard view is taken of requirements for processes, intra-partition communication, and the health monitor. The ESA view is that management of those aspects are handled by a partition guest OS for processes and intra-partition communication, or a system partition for the health monitor. Also, as far as possible drivers are not part of the kernel in order to keep it as simple as it is possible. Only clock drivers are part of the kernel since they are necessary for the partition scheduler. A system partition is responsible for allocating drivers for other devices. User partitions cannot have direct access to devices, and any communication between a device and a user partition is through an explicit inter-partition communication channel between the user partition and the driver system partition.

In the case of security and separation requirements this reference specification follows the SKPP recommendations, and it includes requirements tailored to the space environment for audit, user data protection, identification and authentication, security management, protection of the security functions, and resource utilisation.

Figure 1 shows the diagram for the architectural design. It includes the components required by the reference specification, including cores to manage partitions, communication, global and local time, exceptions not concerned with the kernel security functionality, interrupts and devices, and finally the Kernel

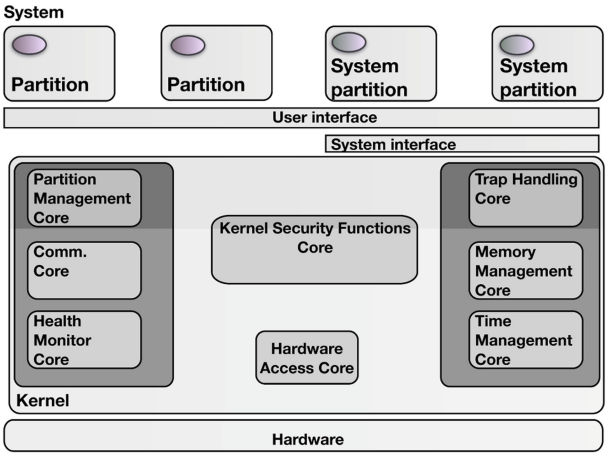

Fig. 1. Separation kernel design

Security Functions (KSF) core. User and System partitions can access the functionality the kernel provides through the interface that these modules implement.

3 Verification Methodology and Toolchain

The complexity of verifying a separation kernel makes it necessary to use verification based on layer abstractions such as refinement.

Our approach to refinement verification is fairly standard, as we take a *forward simulation* approach. We determine a refinement relation $\mathcal{R}_{\mathcal{AI}}$ between the states S_A of an abstract model $\mathcal{M}_A$ and the states S_I of an implementation model $\mathcal{M}_I$. Given $op_I : s_I \mapsto s'_I$ and $op_A : s_A \mapsto s'_A$, showing that op_I data-refines op_A is a matter of showing, given any s_I and s_A, that the diagram in Fig. 2 commutes: Generally, the relation between $\mathcal{R}_{\mathcal{AI}}$ is not trivial, mapping one abstract state to a set of corresponding implementation states. Additionally, in particular when aiming the full kernel verification, data and behavioral simplification of the abstract model bring new invariants in the implementation model that must be verified. So, to simplify the refinement task and the verification, it is typically necessary to introduce intermediate models $\mathcal{M}_{IM_0} \cdots \mathcal{M}_{IM_n}$ with higher levels of abstraction such that $\mathcal{M}_A \sqsubseteq \mathcal{M}_{IM_0} \cdots \mathcal{M}_{IM_n} \sqsubseteq \mathcal{M}_I$.

Although it is well known that refinement preserves safety but not liveness properties [15], in [16] it is shown that it is possible to formulate liveness properties in such a way that they are preserve by refinement.

Our property verification effort will be concentrating on proving system-call correctness and verifying security properties. Proofs for system calls will use classical Hoare triples based on pre- and post-conditions, ensuring that system calls satisfy their functional requirements. With regard to security properties, they will be verified using invariant preservation over data structures related to a security property.

The whole verification process is being carried out using the Isabelle/HOL theorem prover [17]. Isabelle/HOL Higher-Order Language (HOL) allows us to model the kernel abstraction, and to describe the properties representing the security and functional requisites and the refinement relation between models. In order to apply refinement verification we need to obtain an implementation model, which can be obtained using a semantic model of the implementation language over the source code. To that aim, we are using the `C-Parser` tool [18] developed by NICTA (Australia's Information and Communications Technology

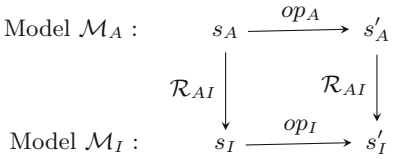

Fig. 2. Refinement (forward simulation)

(ICT) Research Centre of Excellence), that takes as input code a subset of C-99[2], and automatically provides a set of Isabelle/HOL theories defining an imperative model of that code. `C-Parser` introduces proof automation, discharging translation correctness, so the tool ensures that the model is correct with regard the provided semantic model, and additionally it automatically proves properties about memory access correctness.

4 Modelling the Separation Partitioning Kernel

Refinement verification requires the construction of an abstract model of the requirements, simple enough to make the verification feasible. However, it is important to have simplicity in mind as a compromise between verification of requirements and refinement verification, to obtain a transition relation which eases the refinement verification.

The partitioning-separation formal model is composed of more than thirty Isabelle/HOL theories where the kernel architecture and behaviour are defined according to the reference specification. These theories cover the spectrum from low-level machine data-types up to high-level kernel security behaviour. This range of abstract levels is required both to ease the construction of refinement relations, and cover aspects of security that themselves have low-level hardware-related aspects.

Therefore, the abstract model includes elements present in implementations like XtratuM: e.g., partial support for function pointer structures, and features such partitions with two virtual timers (global and local).

4.1 Kernel Data Structures

The kernel global state is modelled as an Isabelle/HOL record containing the current state of the partition-separation kernel components described in Sect. 2, and an additional field representing the current machine state, which is necessary to ensure that setup of hardware (e.g. MMU configuration) enforces data separation.

```
record state=
   partition_manager ::partition_manager_type
   communication_state ::communication_type
   health_monitor :: health_monitor_type
   ksf :: ksf_state
   trap_management :: trap_management_type
```

Collections of objects uniquely identified, like partitions, or communications channels, are specified using partial functions from the field identifying the object (e.g. the partition identifier) to the object itself. Since Isabelle functions are

[2] C-99 refers to the revised standard of ANSI C, or C-89, released in 1999

total, partial functions are specified using datatype *option*, which returns *None* for those identifiers not mapped to any object and *Some obj* for those mapped to object *obj*.

The State and Function Pointers. Operating systems implement trap and interrupt functions using a table mapping traps and interrupts to the function handling them. So, on incoming interrupts or traps, the OS goes to the lookup table and executes the management functions. To model this behaviour on Isabelle/HOL it should be enough to define a function mapping traps and interrupts to higher order functions managing the interrupts. Since traps and interrupt management functions usually modify global variables, it is necessary to model them as functions modifying the state, which can be done using monads.

However, Isabelle/HOL requires that datatype constructors involving the type constructor $\rightarrow$ for the full function space, do not use the newly defined datatype in the $\rightarrow$ lefthandside, i.e. all occurrence must be strictly positive [19]. This is because having a constructor which recursively uses a new type τ on the left side of $\rightarrow$ means that the cardinality of τ would be at least that of the power-set of τ, which by Cantor's theorem would be strictly greater than the cardinality of τ. Therefore, due to this restriction it is not possible to include state modifier functions in the global state, nor in the state of any kernel component.

One work around, is to define the *state* without considering function pointers, and create a new record extending the original state with the set of possible functions being pointed to by these pointers. But that is not enough to allow state components using function pointers, like the machine state, to directly reference those function pointers, since they belong to the *extended_state* definition, which is not visible to them. To solve this, the field extending the *state* containing the functions pointed to by function pointers, is mapped from naturals to higher order functions modifying the state, and function pointer variables keep the natural number associated with the relevant function pointer in the mapping function. Invoking the function pointer means getting the function mapped to that natural. The relevant excerpt from the model is immediately below, noting that we have two different kinds of function pointer: interrupts and trap-handlers. Both take the interrupt or trap being handled and return a monad over a partial state.

```
record extended_state = state + funct_pointers :: funct_pointer_map
type_synonym interrupt_handler = "pointer ⇒ hw_irq ⇒ unit ps_monad"
type_synonym trap_handler = "hw_fault ⇒ unit ps_monad"
datatype funct_pointer = FunctionPointer_1 "trap_handler"
                       |FunctionPointer_2 "interrupt_handler"
type_synonym funct_pointer_map = "funct_pointer_ind ⇀ funct_pointer"
```

However, this work around also has limitations. The most important one is that functions pointed to by function pointers cannot invoke other function pointers. This is because the actual function is a monad to *partial_state*, which

does not include the function pointer mapping. Nonetheless, none of the function pointers modelled are affected by this limitation.

Using *record* related Isabelle/HOL functions is possible to define maps from *state* to *partial_state* and vice versa:

```
definition s_2_ps::"state ⇒ partial_state"
where "s_2_ps s ≡ partial_state.truncate s"
definition ps_2_s::" funct_pointer_map ⇒ partial_state ⇒  state"
where "ps_2_s fp ps ≡  partial_state.extend ps (state.fields fp)"
```

Health Monitor. In the presence of hardware faults, the trap manager sends the trapped fault to the health monitor. The health monitor is modelled as a record containing a fault management table, modelled as a total function from faults to monitoring actions, and a fault record. The fault management table specifies one of the following actions: system restart (cold or warm), shutdown, halt, or ignore. This last action keeps a fault log accessible by the health monitor system partition, as specified by the IMA-SP.

Partition Virtual Machines. As mentioned in Sect. 2 the separation kernel provides a virtual machine to partitions in such a way that they have (apparently) unique access to the hardware. To that aim, in addition to user context registers and the program counters, the partition manager keeps the partition cache status, for both data and code caches, interrupt state, and virtual timers.

Partition cache virtualization keeps the state of cache-enabled and frozen bits for data and cache in the MMU configuration, restoring them on each partition switch from the cache virtualization data of the incoming partition. In addition, on each partition switch the contents of the cache for the outgoing partition is flushed for cache sanitation.

In partition virtualization, interrupt management plays a big role. Fields in charge of interrupt management determine whether virtual interrupts are enabled for the partition and the priority level. Two total functions provide a mapping from hardware and system traps to addresses in the partition trap handler table, with the possibility of masking traps and keeping track of unattended traps. Partitions' trap routines are user code and their execution are beyond the scope of the kernel model. The Isabelel/HOL structure for partition interrupt virtualization is:

```
record trap_info = hw_mask :: trap_mask
   hw_pending :: trap_pending
   ext_mask :: trap_mask
   ext_pending :: trap_pending
   t_enabled :: trap_enabled
   t_PIL :: pil        t_line :: trap_line
   hw_vector :: "hw_irq ⇒ vector_index"
   sys_vector :: "ext_irq + ext_fault⇒ vector_index"
   sw_trap :: "sw_trap option"
```

Machine State. Although the kernel model does not aim at hardware verification, we need a hardware model of those parts which require a correct setup to ensure spatial separation. The machine model depends heavily on the target hardware platform, which in the case of the current project, is the Sparc V8 architecture [20].

In particular, we have modeled the following:

1. *Processor registers* to control supervisor mode and incoming traps and interrupts
2. *User general purpose registers* to check register sanitation on context switches
3. *Input/output devices' address space* to verify correct setup of devices
4. *Memory Management Unit (MMU)* to ensure the directory entries for virtual memory are correctly configured.

(i) and (ii) are modelled by `CPU_context` and `user_context`, defined as total functions from a register datatype, specified as a datatype providing a constructor for each register, to a machine word, specified as a 32-bit word. (iii) is modelled by `io_memory` as a partial function, from the set of devices memory addresses to machine words. Finally `MMU_state` models (iv) and contains the MMU registers, modeled also as total functions from MMU register name to machine words, the data and code cache, and the memory.

4.2 Kernel Behaviour

The separation kernel is an event-oriented model. Indeed, partitions are run owning the microprocessor until some event raises a trap, to wake up the kernel, which will handle the event. Conforming to this, the top Isabelle/HOL theory in the modelled kernel is an entry point waiting for incoming events, i.e. traps and partition calls to kernel services. In addition, as we are concerned with spatial separation verification, memory access events are also considered. With a correct MMU set-up, a partition accessing virtual memory addresses not assigned to that partition shall cause a memory access error.

The kernel states are shown in Fig. 3. The entry point only handles traps, system calls and access to memory when the kernel is in NORMAL or MAINTENANCE states, these being software events ignored in any other state. The BOOT state is partially modelled by a booting function returning an initial kernel state for the given initial configuration.

Traps, Interrupts, and Health Monitor. The model captures hardware traps and device interrupts with a partial function from traps and interrupts to the function handling them. By default, the health monitor handles kernel traps, but partition traps are delivered to a system partition, which carries out health monitor functions. Whether an incoming trap is delivered to the system partition is decided by a configuration table with the actions to take on traps. Interrupts and traps are considered non-preemptive.

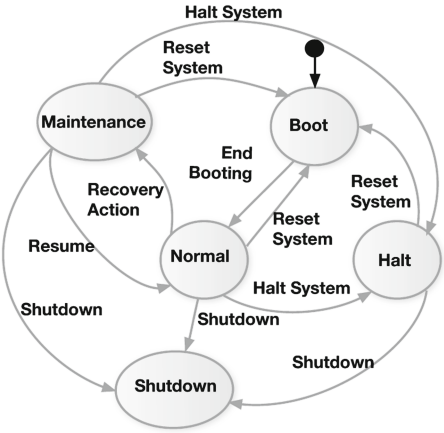

Fig. 3. Kernel state transition graph

Similarly to traps, device interrupts are also handled by a system partition. When a device interrupts the kernel it sets the bit corresponding to that interrupt in the partition virtual machine for the system partition handling interrupts. The kernel only provides handlers for the hardware timers. In particular it provides two handlers: one is for the system time, incrementing the system clock and the running partition system clock; the other is for system timers, which control the partition scheduler and virtual timers for partitions.

The Security Model. The SKPP [4] functional security requirements include: security audit, user data protection, identification and authentication, security management, self-protection, and resource utilization. The most important parts from the temporal and spatial isolation perspective are those defining the flow control functions, and temporal quotas.

The flow control function is modelled in the Kernel Security Function (KSF) module as a function mapping partitions and resources to the allowed operations of partitions over the resources, following a flow control policy based on the Least Privilege abstraction [21] (LP). The LP function defines, for each pair ($partition \times resource$), acceptance or denial of a given operation ($write/read$ allowed or not). In the case the operation is not specified, and the value for that pair is *unspecified*, then the flow control is based on a function mapping operations between partitions, following in that case a partition-pair abstraction. Here the partition pair function PP defines, for each pair ($partition \times partition$), the acceptance or denial of a given operation, and in the case that the value is *unspecified* any operation involving those partitions is forbidden.

The IMA-SP specification [13] forbids shared memory, and the only flow between partitions is using inter-partition communication channels, which communicate via ports uniquely assigned to partitions. Communication channels fix the direction of the communication so each channel has to be composed of a

source port, which writes to the corresponding destination port, ensuring a one way communication flow. Considering inter partition communication, the flow control function model shall allow an operation *op* between partitions $P1$, $P2$ using a inter partition channel $ch = \langle sp \times dp \rangle$, with sp a source port in $P1$, dp a destination port in $P2$, if $LP\ P1\ dp = Write\_Allowed \vee (LP\ P1\ dp = Undefined \wedge PP\ P1\ P2 = Write\_Allowed) \wedge LP\ P2\ sp = Read\_Allowed \vee (LP\ P2\ sp = Undefined \wedge PP\ P2\ P1 = Read\_Allowed)$.

In the case of multi-cast channels with one source port sp and multiple destination ports $dp_1, \ldots, dp_n$, the operation will be allowed if for all pairs $(sp \times dp_1) \ldots (sp \times dp_n)$ the relation above holds.

For the KSF to enforce temporal quotas, it manages the global clock and the set of Major Time Frames (MTF) defining the scheduler. It is in charge of switching partitions, detecting if some partition has exceed its temporal quota and ensuring that residual information is removed. The reason that a partition can exceed its temporal quota is because a partition can invoke a system service just before the current PTW finishes and, since the kernel is not preemptive, the timer interrupt would not be handled until that call finishes. This could be solved by allowing preemption, but making the kernel more complex, or establishing a time in the MTF after which system calls are not allowed.

With regard to security, the KSF models authorization access to kernel privileged operations. The KSF records, for each partition, a set of authorized operations. For the sake of kernel simplicity, system partitions own authorization to all privileged operations, whereas non-system partitions own no rights.

Kernel Services. Kernel services typically change the kernel state into another one modifying some kernel variables, so they are modelled using state monads. The kernel offers two types of services: privileged services like changing the state of a partition or halting the kernel, and non-privileged services such as sending a message to a communication port, or modifying some partition virtual machine property like enabling the data cache. Before invoking any kernel service, the kernel checks out the current partition privileges to detect if the current partition is authorized to perform the operation (check phase). Also, the service arguments are examined to check they are correct (decode phase). Only if the partition has the necessary privileges and the arguments passed to the service are correct the service is invoked (invocation phase), otherwise the kernel will return the corresponding error code.

4.3 The Abstract Security Model

We represent the separation kernel concept with an abstract security model (ASM, Fig. 4) whose configuration is composed of the tuple $< P, \alpha, \beta, \sigma >$, where P is a set of partition-ids, α is an action policy representing a set of allowed actions for each partition-id, β is a set of applications as a function from partitions to sequence of actions, and σ is a schedule, modelled as a sequence of partition-ids, each occurrence denoting the corresponding partition performing one (the next) action from its corresponding sequence.

$$
\begin{aligned}
run &: Sys \to RunHist \\
run(\alpha, \sigma, \beta) &= execute_\alpha(\sigma, \beta) \\
execute &: ActPol \to (Sched \times Apps) \to RunHist \\
execute_\alpha(\langle\rangle, \_) &= \langle\rangle \\
execute_\alpha(p : \sigma, \beta) &= perform^{\beta,p}_{\alpha,\sigma}\beta(p) \\
\\
perform &: (ActPol \times Sched) \to (Apps \times PId) \to Run \to RunHist \\
perform^{\beta,p}_{\alpha,\sigma}\langle\rangle &= execute_\alpha(\sigma, \beta) \\
perform^{\beta,p}_{\alpha,\sigma}(a : r) &= \begin{cases} (p, a) : execute_\alpha(\sigma, \beta \dagger \{p \mapsto r\}) & , a \in \alpha(p) \\ \langle\rangle & , a \notin \alpha(p) \end{cases}
\end{aligned}
$$

Fig. 4. Abstract security model

We say that a run-history run, defined as a sequence of pairs $(Pid \times actions)^*$, is consistent with an action policy α, $hpConsistent_\alpha\ run$, if for every partition/action pair $(p, a) \in run$, α allows p to perform a.

The key property in ASM is that the execution of actions for each scheduled partition, given by $run(\alpha, \sigma, \beta)$, is consistent with the policy: $hpConsistent_\alpha\ run(\alpha, \sigma, \beta)$.

Run takes as input the action policy (α), the scheduler (σ), and the sequence of actions that partitions execute (β), and it returns the run-history. If the sequence of partitions defined by the scheduler is empty, run finishes; otherwise if it is equal to $p : \sigma$ (: is the list constructor) it adds the pair (p, a) to the run history, where a is the next action in the sequence of actions for partition p, $\beta(p)$, and a is taken out from the list of actions for $\beta(p)$ († is the map override operator). If there is no next action a for p, $\beta(p) = \langle\rangle$, the next partition in σ is scheduled and the run history is not modified. If action a is not allowed for partition p, expressed as $a \notin \alpha(p)$, then we simply stop, and return an empty history.

4.4 Refinement Relation between ASM and the Reference Model

For refinement verification (Fig. 5) we define the relation $\mathcal{R}^{ASM}$ between states belonging to ASM and the abstract model of the reference specification (rm). This relates the sequence of actions for partitions β, the scheduler σ, and the action policy α, with the set of partitions defined in the partition manager, the scheduler, and the security policy, respectively.

Function f used in $\mathcal{R}^P$ *and* $\mathcal{R}^{SP}$ defines a function from partition-ids in the very abstract model to partition-ids in the abstract model. Function h associates the partition sequence of actions in ASM with segment codes in the abstract model. Similarly, g associates in $\mathcal{R}^P$ the partition's set of allowed action with the set of allowed operations in the partition security model. For $\mathcal{R}^S$, a prefix p^* of σ must be equal to the partition window ($PTWid$) in the abstract model's

$$\mathcal{R}^{ASM}\ (\beta, \sigma, \alpha)\ rm \equiv\ \mathcal{R}^{P}\ \beta\ rm.partition\_manager \wedge \mathcal{R}^{S}\ \sigma\ rm.scheduler \wedge \mathcal{R}^{SP}\ \alpha\ rm.security\_model$$

$$\mathcal{R}^{P}\beta\ pm \equiv \exists f, h. Dom\ \beta = f`(Dom\ pm.partitions) \wedge \forall i \in Dom\ \beta.\beta\ i = h(pm.partitions(f\ i))$$

$$\mathcal{R}^{S}\ \sigma\ sch \equiv\ \sigma = p^{*} : \sigma_1 \wedge partition\ sch[sch.current][sch.PTWid] = p \wedge \exists pv.\sigma_1 = pv\ sch$$

$$\mathcal{R}^{SP}\ \alpha\ sp \equiv \exists f, g.\ \ Dom\ \alpha = f`(Dom\ sp) \wedge \forall i \in Dom\ \alpha.\alpha\ i = g(pm.partitions(f\ i))$$

Fig. 5. Abstract security model refinement relation

current scheduler *sch.current*. ‘ represents set image function, and : the concatenation operator. Variable *pv* is a prophecy variable [15] introduced to simulate non-deterministic changes of the current scheduler when a partition invokes the corresponding system call as defined by IMA-SP.

5 Towards Separation Kernel Verification: The Xtratum Case

The XtratuM kernel [8] is a general partitioning microkernel, which provides basic hardware access to allocated partitions and is in compliance with the IMA-SP reference specification [13]. XtratuM provides temporal and spatial separation, using specific hardware to access memory and a deterministic cyclic scheduler to provide partitions with spatial and temporal isolation, inter-partition communication, and health monitoring for partition fault management.

5.1 Getting an Implementation Model

To get the implementation model, we used Xtratum 3, release 33, dated June 2012 and we modified it to make it compliant with Nicta's `C-Parser` 1.13, dated May 2013. Xtratum, version 3-33, is composed of more than 10 K. Lines of C code and 440 functions. Therefore, although it is really small in comparison with general purpose kernels (e.g., Linux v3 kernel has more than 19,000 K LoC) it is still big enough to necessitate that the implementation model is obtained automatically.

However, the `C-Parser` tool imposes some restrictions due to verification decisions and parsing considerations. The most significant ones are: function calls and assignments are not considered statements, but expressions, so side-effect expressions are not allowed such as assignments in control flow conditions; to simplify the memory model, pointers referencing local variables are not allowed; union and bitfields are considered unsafe since they violate the abstraction of the C semantic model and hence are forbidden.

XtratuM makes use of a subset of C not covered by `C-Parser`, so it has been necessary to edit XtratuM's source code to make it compliant with the

parsing tool. With that aim, side-effect expressions where split using temporary variables to store the side-effect modification, and referenced local variables were moved out of the local context to the global kernel context. Unions and bitfields have been manually transformed into arrays of bytes of the size of the largest union field. C operators accessing union fields are substituted with functions that access the field in the resulting array of bytes. Note that currently those modifications are implemented to make the use of C-Parser possible, but this means we need to verify of the correctness of our modifications. Nicta uses a specific tool to automate this transformation and provide a translation proof, but as of this time it is not publicly available.

In addition, it is necessary to modify the XtratuM source code to find workarounds to bugs or undocumented C statements not handled by the parser, such as C comments at the end of a line using "`//`", or the `volatile` type qualifier. The above issues are resolved by their removal from the source code because they do not affect the functional behaviour of the implementation. Other problems are resolved by replacement with equivalent but compliant code. While deriving the partial implementation model, we found 14 different incompatibilities between the XtratuM code and the parsing tool.

So far, we have partially extracted an implementation model for XtratuM, having extracted the kernel data definitions and the scheduler functions, and those functions the on which scheduler depends. To obtain a complete model using `C-Parser`, it is necessary to restructure the source code and to provide a model of the kernel's assembly code, which the parser ignores. First, as is usual in a Kernel's code organization, XtratuM is organized in different modules which are independently compiled, and later linked into the final executable file. In this scheme, inter-module dependencies are resolved by exporting function prototypes that other modules use. It is common to find cyclic dependencies between modules which are resolved during the linking stage. On the other hand, `C-Parser`'s input is a unique C file, where such cyclic dependencies are not allowed. Hence, it was necessary to carefully re-structure the source code to be compatible with `C-Parser`. In future we may also explore the use of the CIL tool (http://sourceforge.net/projects/cil/) to merge all the C files into one without such cycles. Indeed, we are getting a modified kernel where formal verification ensures its correctness and that its behaves as expected by the specification. Second, kernels use assembly code in their lower layers to access the hardware, especially for functionality concerning interrupt management and hardware configuration. Although at this stage it is not intended to perform machine code verification, it is at least necessary to provide a minimum model to ensure hardware configuration correctness, especially of critical elements for supporting isolation such as the MMU.

5.2 Refinement Model

The construction of the abstract model was developed considering both the reference specification [14] and the XtratuM implementation foreseeing the refinement relation with the implementation model. Nevertheless, and although both

models have a similar architecture, some of the data structures in the implementation differ dramatically from the abstract model, therefore the refinement relation is not straightforward. This may have a negative impact on the cost of the refinement proof. In particular, most of the data structures in the abstract model are represented as partial functions, while the implementation uses static arrays or linked lists, and often the implementation splits the data structures into multiple arrays or lists, aiming for efficiency, but requiring us to check that the partial functions in the abstraction truly correspond with the implementation.

It is worth noting, that although the refinement relation must connect the whole model with the implementation model, there are some components like the KSF's security policy that are not currently refined since the implementation does not provide an explicit model of it, although there is a functional implementation of it implicit in inter-partition communication.

6 Future Work

Verifying a kernel is a major task, with complete verification requiring several years of effort, even in the case of a small separation kernel consisting of only a few thousands of lines of code. After finishing the abstract model that spans functional and security requirements, there are still two big steps to carry out: Requirement verification over the abstract models and proof of refinement between the abstract model and the implementation model.

Although verification of requirements can be considered a conceptually straightforward task, albeit with a lot of attention to a lot of detail, mostly focused on invariants and Hoare-triplet verification, it is still necessary to bring forth a proof of separation to support a guarantee of partition isolation in this architecture.

The proof of refinement will require collaboration with the XtratuM team which will help to produce a modified `C-Parser` compliant kernel. Additionally it may be necessary to modify the model extractor to be a better fit to XtratuM, or other alternative implementations, with particular features (e.g., total parser support for function pointers). Moreover, the refinement verification could require an intermediate layer to cover the gap between the abstract model and the implementation model.

What is very clear, is that an approach that requires verifying a refinement relation between independently developed specifications and implementations, is more complex than one where one end of the refinement relation was developed with the other end already known and understood. The MTOBSE project was looking at the situation where a customer issues the specification, and then seeks suppliers to tender their implementations, in open competition — a situation very common were customers are tax-payer/government funded entities such as ESA.

References

1. Rushby, J.M.: Design and verification of secure systems. SIGOPS Oper. Syst. Rev. **15**, 12–21 (1981)
2. Alves-Foss, J., Oman, P.W., Taylor, C., Harrison, S.: The MILS architecture for high-assurance embedded systems. IJES **2**(3/4), 239–247 (2006)
3. Criteria, C.: Common criteria for information technology security evaluation, version 2.3. Technical report ISO/IEC 15408:2005, Common Criteria (2005)
4. Directorate, I.A.: Protection profile for separation kernels in environments requiring high robustness. Technical report, U.S. Government, June 2007
5. ARINC, Arinc specification 653–2: Avionics application software standard interface part 1 - required services. Technical report, Aeronautical Radio INC (2005)
6. Baumann, C., Beckert, B., Blasum, H., Bormer, T.: Better avionics software reliability by code verification a glance at code verification methodology in the verisoft xt project. In: Embedded World 2009 Conference (2009)
7. Klein, G., et al.: seL4: formal verification of an OS kernel. CACM **53**, 107–115 (2010)
8. Crespo, A., Ripoll, I., Masmano, M.: Partitioned embedded architecture based on hypervisor: The xtratum approach. In: Eighth European Dependable Computing Conference, EDCC-8 2010, Valencia, Spain, 28–30 April, pp. 67–72. IEEE Computer Society (2010)
9. Consortium, A.-I.: AIR II (2013). http://air.di.fc.ul.pt/air-ii/. Accessed 5 Nov 2013
10. Heitmeyer, C., Archer, M., Leonard, E., McLean, J.: Applying formal methods to a certifiably secure software system. IEEE Trans. Softw. Eng. **34**, 82–98 (2008)
11. Dam, M., Guanciale, R., Khakpour, N., Nemati, H., Schwarz, O.: Formal verification of information flow security for a simple arm-based separation kernel. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & #38; Communications Security, CCS '13, pp. 223–234 (2013)
12. de Amorim, A.A., Collins, N., DeHon, A., Demange, D., Hritcu, C., Pichardie, D., Pierce, B.C., Pollack, R., Tolmach, A.: A verified information-flow architecture. In: 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL) (2014)
13. de Ferluc, R.: Report D10 – TSP Services Specification. IMA-SP/D10 Issue 2.1, ESTEC. Contract ESTEC 4000100764, March 2012
14. Butterfield, A., Sanan, D.: Reference specification for a partitioning kernel and a separation kernel. Technical Report Version 4.1, Lero. ESTEC Contract No. 4000106016, 3 parts: MTOBSE_D03i4-SRS,-ICD,-ADD (2013)
15. Abadi, M., Lamport, L.: The existence of refinement mappings. Theor. Comput. Sci. **82**, 253–284 (1991)
16. Alves-Foss, J., Rinker, B., Benke, M., Marshall, J., O'Connel, P., Taylor, C.: The idaho partitioning machine. Technical report (2002)
17. Paulson, L.C. (ed.): Isabelle: A Generic Theorem Prover. LNCS, vol. 828. Springer, Heidelberg (1994)
18. Winwood, S., Klein, G., Sewell, T., Andronick, J., Cock, D., Norrish, M.: Mind the gap: a verification framework for low-level C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 500–515. Springer, Heidelberg (2009)
19. Berghofer, S., Wenzel, M.: Inductive datatypes in HOL - lessons learned in formal-logic engineering. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L. (eds.) TPHOLs 1999. LNCS, vol. 1690, p. 19. Springer, Heidelberg (1999)

20. C. SPARC International Inc., The SPARC architecture manual: version 8. Upper Saddle River, NJ, USA: Prentice-Hall Inc. (1992)
21. Saltzer, J.H., Schroeder, M.D.: The protection of information in computer systems. Proc. IEEE **63**, 1278–1308 (1975)

MARKED PROOF

Please correct and return this set

Please use the proof correction marks shown below for all alterations and corrections. If you wish to return your proof by fax you should ensure that all amendments are written clearly in dark ink and are made well within the page margins.

Instruction to printer	*Textual mark*	*Marginal mark*
Leave unchanged	··· under matter to remain	New matter followed by ⌐ or ⌐(x2)
Insert in text the matter indicated in the margin	⌐	
Delete	/ through single character, rule or underline or ⊢⊣ through all characters to be deleted	⌐/ or ⌐/(x2)
Substitute character or substitute part of one or more word(s)	/ through letter or ⊢⊣ through characters	new character / or new characters /
Change to italics	— under matter to be changed	⌐
Change to capitals	≡ under matter to be changed	≡
Change to small capitals	= under matter to be changed	=
Change to bold type	~ under matter to be changed	~
Change to bold italic	≈ under matter to be changed	≈
Change to lower case	Encircle matter to be changed	≠
Change italic to upright type	(As above)	⌐
Change bold to non-bold type	(As above)	⌐
Insert 'superior' character	/ through character or ⌐ where required	⌐ or ⌐ under character e.g. ³⌐ or ³⌐
Insert 'inferior' character	(As above)	⌐ over character e.g. ⌐₂
Insert full stop	(As above)	⊙
Insert comma	(As above)	,
Insert single quotation marks	(As above)	⌐ or ⌐ and/or ⌐ or ⌐
Insert double quotation marks	(As above)	⌐ or ⌐ and/or ⌐ or ⌐
Insert hyphen	(As above)	⊢
Start new paragraph	⌐	⌐
No new paragraph	⌐	⌐
Transpose	⌐	⌐
Close up	linking ⌒ characters	⌒
Insert or substitute space between characters or words	/ through character or ⌐ where required	⌐
Reduce space between characters or words	⌐ between characters or words affected	⌐