

Using Isabelle/HOL to Develop and Maintain Separation Invariants for an Operating System (Extended Abstract)

Holger Blasum¹, Oto Havle¹, Bruno Langenstein², Yakoub Nemouchi³, Julien Schmaltz^{4,5},
Werner Stephan², Sergey Tverdyshev¹, Freek Verbeek⁴, and Burkhart Wolff³

¹ SYSGO AG,

² DFKI GmbH,

³ Université Paris-Sud,

⁴ Open Universiteit Nederland,

⁵ Eindhoven University of Technology

Abstract. We report on the work on formally verifying PikeOS – an industrial separation kernel – in the Isabelle/HOL theorem prover. Our PikeOS model basically specifies the behavior of different API calls such as IPC, events or ports. This work poses a proof engineering challenge: how can we create a realistic model of PikeOS with a state that is detailed and comprehensive, while maintaining manageability of all proofs? Our current solution makes heavy use of extensible records to achieve modularity. Ultimately, this allows us to derive a proof of security for the entire separation kernel out of local proofs done per API call.

1 Introduction

As described on the EURO-MILS website (<http://www.euromils.eu>), the main goals of the EURO-MILS project are to develop market-relevant technologies and concepts for virtualization of heterogeneous (embedded) systems and the formal verification for those systems as part of rigorous cross-European security certification.

In particular EURO-MILS focuses on verifying PikeOS, an industrial separation kernel [2,1]. A separation kernel is a special kind of operating system, that allows to effectively separate different applications running on the same platform from each other. An application is provided by an arbitrary user as arbitrary binary code. In particular, this definition of user application includes virtualization: the separation kernel can host user applications that can also be operating systems. User applications also can be malicious, and even in that case the separation kernel ensures that malicious user applications are harming neither the separation kernel nor other applications in other partitions. The separation kernel will be installed and run on a hardware platform (e.g. embedded systems). Thus the most interesting property of a separation kernel is that it enforces the separation it promises.

To verify that PikeOS provides separation, we are working on 1.) a generic model of separation kernels, 2.) a model of PikeOS and 3.) showing that the model of PikeOS is an instantiation of the generic model. In the remainder of this paper, we give a high-level snapshot of ongoing development of these proof efforts. We focus on a specific part of the project that turned out to be an Isabelle proof engineering challenge, namely, how to model the state of the PikeOS instantiation in such a way that discharging the proof obligation remains manageable. To this end, we made heavy use of extensible records [4].

2 A Generic model of Separation Kernels

For showing separation, we use the separation framework by Rushby [5]. Rushby models a separation kernel as a Mealy machine. He then defines a number of proof obligations and proves that any Mealy machine that satisfies these proof obligations enforces separation. For EURO-MILS, we have refined the Rushby separation framework. Instead of an abstract Mealy machine, we have a generic model that includes interrupts, sequences, concurrency and multicore to a “Controlled, Interruptible Separation Kernel” (CISK) in [7,6]. For this model we have proven separation. Necessarily, this required additional proof obligations.

2.1 Preemptible and Abortable Execution

An operating system has the task of managing resources and providing services (system calls) to applications. To keep response times short, many system calls are interruptible, i.e., they consist

of (potentially many) non-interruptible code sections and interruptible code sections. Interruptible means that after an interruption, the thread could be scheduled away. Non-interruptible code sections are also called atomic.

When a security mechanism such as an access control is contained in an atomic section (a single look-up of data), a useful abstraction of a system call is a sequence of atomic actions (labeled “A” in Figure 1), with a preemption points (“P”) between each two subsequent atomic actions (points where interrupts and rescheduling may happen).

[A]; [A, P, A]; [A, P, A, P, A]

Fig. 1. Action Sequence

It is up to the security model to provide a meaningful and human-readable abstraction so atomic actions are modeled if and only if they implement some security check or conversely provide some mechanism that is considered relevant for the security of the system (e.g., change of an access control matrix).

In particular a system call typically has a structure that at each preemption point is either successful (it may continue towards the next interruption point) or it fails (it aborts), or thirdly, it goes into a state where it is waiting for some atomic action from another thread. We capture this behavior by specifying maximal sequences of atomic actions, and for each sequence, separately, its aborting and its waiting behavior. In terms of Figure 1, a maximal sequence only explicitly lists atomic actions (“A”), the always-existing preemption points (“P”) in between are implicit.

Example 1. Consider, e.g., the specification of a maximal sequence [X-Check, X-Do]. This means that the following set of sequences can happen [X-Check], [X-Check, X-Do]. When the maximal sequences [X-Check, X-Do] has degenerated to [X-Check] it means that the access control check [X-Check] failed and [X-Do] is not done. In the implementation, when a thread that has passed [X-Check] has succeeded is interrupted, the result of [X-Check] is kept implicitly in the program counter of the interrupted check, which is pointing to a code location only reachable by successfully modelling the check. In the model, when [X-Check] has succeeded we have to keep this result somewhere else. We choose a “precondition” predicate for this. This is similar to a narrative description “you can do [X-Do] when you have passed [X-Check]”.

In the rest of this paper, we use the term “action” as shorthand for “atomic action”. The step function consists of either

- an execution of an action, or
- the aborting of an action, or
- the delaying of an action, or
- a context switch executed by a scheduler.

Related actions, i.e., actions that operate on the same data, are structured into *action groups*. Per action group, 22 proof obligations have to be discharged.

2.2 Some Proof Obligations

Many proof obligations are based on a *view partitioning equivalence* (vpeq). This predicate is an auxiliary construction for proofs. View partition equivalence expresses that two states look equivalent from the view of a subject.

Local Respect The system locally respects the security policy P if it ensures that whenever the domain $u1$ currently active in state s cannot influence domain $u2$, the state s is vpeq from the point of view of the subject $u2$ to the state obtained after doing the action. In practice, this proof is done as follows. Assume P forbids that $u1$ influences $u2$. Then by unfolding the definition of the action the state s as seen by $u2$ remains unchanged.

Weak Step Consistency Weak step consistency ensures that two $u1$ -vpeq states s and t that are also a -vpeq remain $u1$ -vpeq after executing an action a , that is a call to the step function. We assume that a precondition for doing action a in states s and t holds. Also, we assume that the same domain is active in states s and t .

3 A Model of PikeOS

We model PikeOS by representing its kernel API operations. These API operations can be called in the code of user applications (note that each API call corresponds to a CISK action group). Each API operation call makes use of a particular part of the state. For each API call, we specify:

- the data structures within the state that are *used* by that call
- an abstraction of how the API call operates on the relevant part of the state
- when the action reaches a preemption point (its “waiting” predicate)
- under which conditions the action is aborted.

Each additional data structure in the state is a refinement of the abstract model. Isabelle’s concept of *extensible records* is used to structure the kernel state. For example, data structures for events (event counters) and IPC (IPC states) are added to the (version of) the state type in theories where we show correctness of event and IPC invocations. Ultimately, this state type is extended to a version with several components addressing diverse features of the PikeOS Kernel. All these constituents are modelled as record extensions.

current. The state stores which user-application is currently active.

policy. The communication rights for all subjects are stored in the state, so that the security policy can be updated during run-time. As our applications are arbitrary we assume them to be intending to achieve maximal communication rights possible during a run-time. However, the run-time behavior of even malicious applications is bounded by a maximal configuration-time static policy.

memory. The system contains several objects such as files, ROM and sampling ports.

API specific data structures. As explained, each API call adds its own data structures to the state.

The different API calls that currently have been modelled are: IPC calls, events, ports and files. For each of these action groups, we have discharged proof obligations such as `local_respect` and `weak_step_consistent` under preemptible and abortable execution. The proofs are simplified by observing that groups of API calls operate on well-defined data, and by grouping proofs on these API calls together. Figure 2 provides an overview. On the X-axis, different constituents of the state are shown, while different action groups are shown on the Y-axis.

State constituents API calls	current	policy	memory	event	IPC
Policy invariant vpeq		x x			
FILES/PORT vpeq step prec/abort/wait			x x x		
EVENT vpeq step prec/abort/wait		x		x x x	
IPC invariant vpeq step prec/abort/wait		x x	 x x		x x x x

Fig. 2. Overview of State Constituents. The ‘x’ marks lemmas that show dependence.

The table shows that any API call is independent of the part of the state that stores which user-application is currently active. As such, user-applications cannot influence the timing-schedule,

which is an important property for separation. The crosses on the diagonal are necessary: an API call depends on the data structures introduced for that API call. The upper-right triangle is necessarily empty: the event API, e.g., works on a state type version that is not extended with the IPC data structures. It is therefore implicitly independent of that part of the total state. The lower-left triangle should be as empty as possible. If it would have been completely empty, then all API calls could be verified completely independently and from that we could derive a total proof of separation. However, certain dependencies occur. For example, whether an event API call can be aborted depends on the part of the state that stores the communication rights.

We illustrate the discharging of proof obligations per API call by discussing, as an example, proofs on the event API.

4 Example: Proving CISK Proof Obligations for the Event Action Group

We give an example for proving the CISK proof obligations for a theory that represents part of the API, namely the *event*. The event API provides the following functionality:

- a thread can emit a signal to another thread by increasing an event counter (`ev_signal`)
- a thread can enter into a wait state, where it is waiting for a signal. When the signal arrives, the thread can either decrement the event counter by one (`ev_wait_one`) or set it to zero (`ev_wait_all`).

When an API call can be interrupted, giving a maximal action sequence consisting of more than one atomic action, it is specified in terms of individual atomic actions. Otherwise, the entire API call is treated as one atomic action.

Each atomic action is described in the following terms:

aborting: the condition under which the action is aborted,

waiting: the condition under which the thread enters a wait state (waiting for another thread),

state change: the state change the thread effects if it does not abort or wait

output: what is output by the action

precondition: precondition of an atomic action; this depends on its position inside an atomic action sequence

Figure 3 gives the structure of the proof argument for showing CISK proof obligations. The proof organization has been organized into files as follows: Lemmas that have been proven within a local Isabelle theory file (`step_ev.thy`) are prefixed by “`ev_`”, called *event-local theory* below. Lemmas that have been proven in a global proof integration file are *not* prefixed by “`ev_`”. When during proof integration, a lemma has been used multiple times, the arrow is given as dashed line.

4.1 Proof Arguments within the Event-Local Theory

The proof arguments made within the event-local theory can be described the following way:

Auxiliary lemmas for information and control flow

- **ev_step_dependence** (information flow): proves that only a limited portion of the state is modified by any event action
- **ev_vpeq_dependence** (information flow): establishes that only a limited state that is read by any event action
- **ev_pred_dependence** (control flow): shows that aborting, waiting and precondition depend only on limited state which is read by any event action

Lemmas directly corresponding to with CISK proof obligations

- Consistency arguments: Consistency arguments would be violated if an action call would rely on information that was not made available before the action.
 - Weak step consistency of the action itself (`ev_weakly_step_consistent`)
 - Consistency arguments on waiting only depend on the waiting behavior (`ev_waiting_consistent`)
 - Consistency arguments on aborting only depend on the aborting behavior (`ev_aborting_consistent`)
- Local respect (`ev_local_respect`) arguments: it has been proven globally in CISK that aborting invocations does not violate local respect. Therefore, local respect arguments proceed as follows:
 - assert that any dynamic state of the access control policy is bounded by a static state of the access control policy,
 - assert from non-aborting that the access control policy is not violated,
 - from this infer view-partition equivalence of the action.

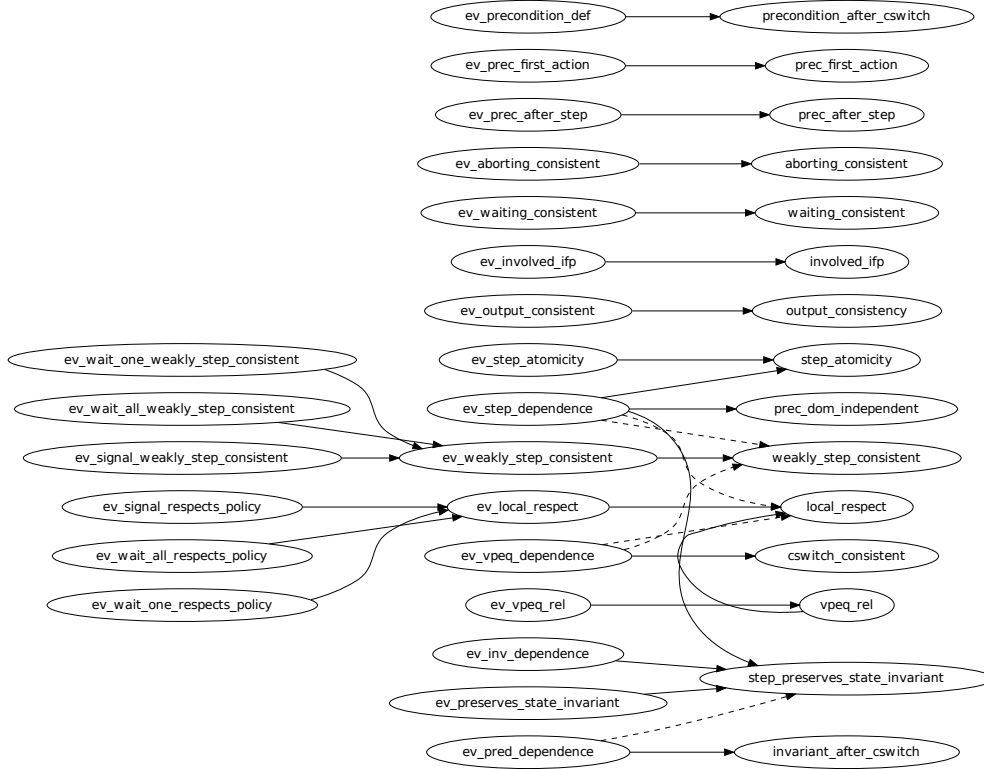


Fig. 3. Structure of arguments for showing CISK proof obligations for the event action group

4.2 Interfacing with CISK by Proof Integration

- **weakly_step_consistent** is established by a case split on each action group (proof by case distinction), in our case for the event action group. The argument for the case that the action is an event from the event action group comprises:
 - use **ev_step_dependence** to assert that dynamic policies are not changed,
 - show that the state other action groups are acting on is not affected by event action groups (using ***_vpeq_dependence**, except for **ev_vpeq_dependence** that is only used for the *other* cases), and
 - show weak step consistency of the event action itself. This is where **ev_weakly_step_consistent** is used.
- **local_respect** is also done for each action group (proof by case distinction), in our case for each event action group. The argument for the case that action is an event from the event action group comprises:
 - use **ev_step_dependence** to assert that dynamic policies are not changed,
 - a proof that the state other action groups are acting on is not affected by this event action (using ***_vpeq_dependence**, except for **ev_vpeq_dependence** that is only used for the *other* cases), and
 - a proof of weak step consistency of the event action itself, this is where **ev_local_respect** is used.
- **step_preserves_state_invariant** is again a proof by a case distinction for each action group
 - for each other action group, ***_pred_dependence** holds, and **ev_preserves_state_invariant**, **ev_step_dependence**, and **ev_inv_dependence** hold, derive that state invariant is maintained

4.3 Representativeness of the Event Action Group

The event-local API is characterized by the property that its state changes are clearly restricted to event counters, which cannot be accessed by other API invocations. The situation is more complicated with IPC system operations which can dynamically change the sharing between virtual memory, which is required for efficient data exchange between tasks.

The IPC does not only depend on a part of an IPC specific state, but also on the virtual memory state. The virtual memory is changed by memory actions and file actions. Memory invariants and memory-vpeq-related properties must be proved for all these step functions. There is some duplication in the proofs about virtual memory changes, although there are some reused lemmas in `step_mem` (e.g. `mem_update_preserves_mem_invariant`).

5 Results and Discussion

We have given a snapshot of some of the ongoing developments in the EURO-MILS project. As this is still work in progress, the structure of the proofs is still open for debate.

As related work, Rushby introduced the concept of reference monitor and, in terms of proof obligations, reference monitor assumptions for structuring proof obligations [5]. In the event-local theory, our auxiliary lemmas `ev_step_dependence` and `ev_vpeq_dependence` on information flow appear similar to these reference monitor assumptions.

The most prominent example of a verified separation kernel is seL4 [3]. Their work is bottom-up in the sense that a full proof at the most detailed level possible has been done (as far as we know the proof is not public, nor is their system preemptible, so we cannot simply reuse it). Conversely, we progress top-down on an existing system. As a result, our approach is possible with limited resources to an arbitrary depth.

This simplification does not come for free: The price of simplifying a system very much always is the danger of having overlooked detail. We are aware that the top-down approach that we are following may “break” under further refinements (or simply informal rewritings of the theories by adding more detail): the refinement paradox. The refinement paradox is fully acknowledged by our approach, it is a search methodology for finding possible inconsistencies. We start with a model that is highly generic. Adding detail may expose violation of separation, in which case one has the options of (1) reviewing the model accuracy, (2) achieving separation by configuration restrictions, (3) achieving separation by implementation changes. The approach is similar to testing, where you also define some high-level property to be asserted and then show that exposing it to test cases does not violate the high-level property.

References

1. Jacques Brygier, Rudolf Fuchsen, and Holger Blasum. PikeOS: Safe and secure virtualization in a separation microkernel. Technical report, SYSGO, 2009.
2. Robert Kaiser and Stephan Wagner. Evolution of the PikeOS microkernel. In *First International Workshop on Microkernels for Embedded Systems*, page 50, 2007.
3. Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. seL4: from general purpose to a proof of information flow enforcement. In *34th IEEE Symposium on Security and Privacy*, 2013. <https://www.ieee-security.org/TC/SP2013/papers/4977a415.pdf>.
4. Wolfgang Naraschewski and Markus Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In *Theorem Proving in Higher Order Logics*, pages 349–366. Springer, 1998. <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.71.7895>.
5. John Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-02, Computer Science Laboratory, SRI International, Menlo Park, December 1992. <http://www.csl.sri.com/papers/csl-92-2/>.
6. Freek Verbeek, Juijen Schmaltz, Sergey Tverdyshev, Holger Blasum, and Oto Havle. CISK: a model for separation kernels. 2014. Submitted for publication.
7. Freek Verbeek, Julien Schmaltz, et al. Formal specification of a generic MILS separation kernel. Technical Report D31.1, EURO-MILS: Secure European Virtualisation for Trustworthy Applications in Critical Domains, FP7/2007-2013,, 2013. <http://www.euromils.eu/index.php/publications>.