

Implicit Assumptions in a Model for Separation Kernels

[Extended Abstract]

Freek Verbeek
Open University of The
Netherlands
Radboud University Nijmegen

Julien Schmaltz
Eindhoven University of
Technology

Sergey Tverdyshev
SYSGO AG

Holger Blasum
SYSGO AG

Oto Havle
SYSGO AG

ABSTRACT

In joint work with several industrial and academic partners throughout Europe, we are working towards the certification of PikeOS – an industrial separation kernel developed at SYSGO – according to the highest level of the Common Criteria. We present a new generic model of separation kernels that includes interrupts, control and context switches. For this generic model, noninterference has been proven in the Isabelle/HOL theorem prover from proof obligations on the step function. Noninterference holds for *all* separation kernels which satisfy these assumptions. In this paper, we discuss this methodology for certification purposes. On the one hand, it is clear that our instantiation of the step function for PikeOS must satisfy the proof obligations. On the other hand, there are many implicit assumptions made in defining the run function, that is, in defining how the step function is applied. Based on work-in-progress, we address the issue of providing evidence that PikeOS indeed satisfies our generic notion of noninterference.

1. INTRODUCTION

Separation kernels are at the heart of many modern security-critical systems [7]. PikeOS is a separation kernel developed at SYSGO [6, 2]. A consortium of several European partners from industry and academia works on the certification of PikeOS up to at least Common Criteria EAL5+, with “+” being applying formal methods compliant to EAL7. Our aim is to derive a precise model of PikeOS and a precise formulation of the PikeOS security policy.

In [11] we present a new generic model for separation kernels called CISK (for: Controlled Interruptible Separation Kernel). This model contains several facets present in many separation kernels, such as *interrupts*, *context switches* between domains and a notion of *control*. Regarding the latter,

this concerns the fact that the kernel exercises control over the executions as performed by the domains. The kernel can, e.g., decide to skip actions of the domains, or abort them halfway. We prove that any instantiation of the model provides intransitive noninterference. The model and proofs have been formalized in Isabelle/HOL.

This suggests that discharging the proof obligations for the PikeOS instantiation is enough. The aim of this paper is to spark a discussion about the fact that proof obligations are *explicit* assumptions and the generic model inherently entails many *implicit* assumptions. To actually be able to deduce that PikeOS is a valid instantiation of CISK, these assumptions must be made explicit and it must be argued that they are indeed satisfied by PikeOS. After introducing the CISK model, we present these implicit assumptions and discuss their validity for PikeOS.

2. CISK

This section presents a simplified version of CISK, a generic model of a Controlled Interruptible Separation Kernel. In this model, the entire system consists of several domains running on top of the kernel. Each domain has its own program code. The program code is modelled as a list of kernel API calls. Each API call can trigger a list of atomic kernel actions. The system is deterministic, but we do not make any assumptions on when interrupts occur or what timing schedule is maintained by the kernel.

2.1 Execution Semantics

The model makes use of the following types. Between brackets are the variable names used to refer to objects of these types. Types preceded by an apostrophe are polymorphic. This means that, e.g., the set of domains represented by the type `'dom_t` is parametric.

`'dom_t` ($d, u, v, w, \dots$)

A domain is an entity executing actions and making calls to the kernel. Later on, we define security at the level of domains.

`'action_t` ($a_0, a_1, \dots$)

Actions represent atomic instructions that are executed by the kernel. As kernel actions are assumed

to be atomic, an interrupt may occur after each kernel action.

'action_t execution ($\alpha_0, \alpha_1, \dots$)

An execution of some domain is the code or the program that is executed by the domain. One call from a domain to the kernel will typically trigger a succession of one or more kernel actions. Therefore, an execution is represented as a *list of lists* of kernel actions. Non-kernel actions are not taken into account.

'state_t ($s, t, \dots$)

The state is assumed to contain information about the resources of the system, as well as which domain is currently active. A state does *not* need to include the executions of the domains, as in our model the actions that are executed are modelled separately.

'output_t

Given the current state and the action that is executed by the current domain, the observable output can be determined.

time_t ($n, n', \dots$)

Time is modelled using natural numbers. Each atomic kernel action can be executed within one time unit.

A *system execution* is a mapping of executions to domains, i.e., it assigns to each domain an execution. We use variables $x_0, x_1, \dots$ to refer to system executions. Figure 1 presents an overview of the types used for modelling the program code of the system.

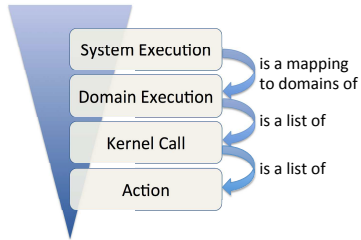


Figure 1: Overview of types

Consider, for example, an IPC call (for: Inter Process Communication). From the point of view of the programmer this is one kernel API call. From the point of view of the kernel it is an action sequence consisting of three stages IPC_PREP, IPC_WAIT, and IPC_SEND. During the PREP stage, it is checked whether the IPC is allowed by the security policy. The WAIT stage is entered if a thread needs to wait for its communication partner. The SEND stage is data transmission. Therefore, the program code is modelled as a list of lists of atomic actions. At each stage, an interrupt can occur, which can give the turn to another domain.

2.2 Isabelle Formalization

We use the Isabelle concept of *locales* to introduce generic functions, i.e., functions without a definition.

locale CISK

fixes kstep :: "'state_t $\Rightarrow$ 'action_t $\Rightarrow$ 'state_t"

and current :: "'state_t $\Rightarrow$ 'dom_t" ("Δ")

and handle :: "'time_t $\Rightarrow$ 'state_t $\Rightarrow$ 'state_t"

and interrupt :: "'time_t $\Rightarrow$ bool"

and aborting :: "'state_t $\Rightarrow$ 'action_t $\Rightarrow$ bool"

and waiting :: "'state_t $\Rightarrow$ 'action_t $\Rightarrow$ bool"

Function *kstep* (for: kernel step) takes as input the state and an atomic action and executes that action. At all times, one domain is active. Function Δ returns that domain given the current state. Interrupt handling is modelled by two generic functions. Function *interrupt* is an oracle that tells when an interrupt will occur. The handler (function *handle*) can then update the state. Function *aborting* and *waiting* take as input the state and an action and decide whether or not the action is aborted or delayed.

On top of these generic functions, we implement a control mechanism *c*. The first task of this control mechanism is to decide, given the current state *s* and the execution α of the current domain, which action to execute. Secondly, function *c* updates the execution. We introduce a datatype that allows function *c* to express whether an action *a* is to be executed (*Do a*), not to be executed (*Dont a*) or whether no action is elected at all (*Nil*).

fun c :: "'state_t $\Rightarrow$ 'action_t execution $\Rightarrow$

('action_t do $\times$ 'action_t execution)"

where "c s [] = (Nil, [])"

| "c s ([#]) = (Nil, [])"

| "c s ([#α']) = (Nil, α')"

| "c s ((a#as)#α') = (if aborting s a then

(Dont a, α')

else if waiting s a then

(Dont a, (a#as)#α')

else

(Do a, as#α'))"

The first two cases occur when the given domain execution α is empty. In the other cases, the domain execution consists of a head (a list of atomic actions representing the current API call) and a tail (the remainder of the domain execution). The third case is when the head is empty, i.e., the execution of the current API call has been finished. Function *c* does not elect an action, but updates α indicating that in the next time slot, the system can continue executing with the

next API call of α . In the final case, the current API call is non-empty, indicating that the kernel is in the process of executing that API call. Three sub-cases can occur. Action a can be aborted. In that case, the entire current API call is aborted and the domain execution is updated accordingly. Action a can be delayed, in which case nothing happens. Finally, function c can also decide to execute action a and proceed as usual.

Using function c we build a run function. We introduce some notation. Function *empty* takes as input a domain execution and returns true if and only if that execution is empty. Functions c_a and c_x denote the first and second output of control mechanism c , i.e., they return the elected action and the updated domain execution.

function run ::

"time_t $\Rightarrow$ 'state_t $\Rightarrow$ ('dom_t $\Rightarrow$ 'action_t execution) $\Rightarrow$

'state_t"

where "run 0 s x = s"

| "interrupt n $\Rightarrow$ run (Suc n) s x = run n (handle n s) x"

| "empty(x (Δ s)) $\Rightarrow$ run (Suc n) s x = run n s x"

| "otherwise $\Rightarrow$

run (Suc n) s x = (let $c_a = c_a$ s x;

$c_x = c_x$ s x in

case c_a of Nil $\Rightarrow$ run n s c_x

| (Dont a) $\Rightarrow$ run n s c_x

| (Do a) $\Rightarrow$ run n (kstep s a) c_x)"

The first case ensures termination, by ending the run function whenever time runs out. The second case deals with interrupts. If at time slot n an interrupt occurs, then no action is executed. Instead, the handler is called. Thirdly, if the current domain has no program code to execute, nothing happens. Otherwise, the control mechanism c is called to decide what needs to happen: either nothing happens, or function *kstep* is called to execute one atomic action.

3. VALIDATION OF CISK

We have proven noninterference over CISK. First, we formulated a set of *proof obligations* over the generic functions. Many of these are similar to those of Rushby (weak step consistency, locally respects, output consistency, etc.) [8]. Using these proof obligations as assumptions, we have formulated an Isabelle proof that information flows only where allowed by the security policy. Formally, this means that any *instantiation* of CISK is secure, i.e., has the property noninterference. This is the top part in Figure 2. The lower part shows the efforts needed to show that an instantiation – in our context, PikeOS – has the property of nonin-

terference. An instantiation is a concrete interpretation of all generic types and functions of CISK. For example, we need to give a concrete definition for the set of PikeOS domains and for the set of actions. Furthermore, a concrete step function must be implemented for PikeOS and concrete definitions that decide whether actions are aborted or delayed. After providing the definitions for the instantiation, all proof obligations have to be discharged. For example, an explicit assumption is that an atomic action a cannot influence which domain is currently active. This assumption has been formalized in Isabelle and therefore – when making an instantiation – Isabelle requires that this proof obligation is discharged. It is then proven – within the Isabelle/HOL theorem proving system – that the step function defined for PikeOS satisfies all proof obligations. It is important to notice that in this proof, PikeOS is modelled as function *run* where generic function *kstep* is replaced with function *pikeos_step*, and functions *aborting*, *waiting*, etc. all have been replaced by their PikeOS counterpart.

The proof obligations can be regarded as a set of *explicit* assumptions that have to be satisfied by the instantiation. However, inherently a generic model also entails many *implicit* assumptions. For example, we assume the system to satisfy non-bypassable, evaluable, always invoked, and tamper-proof (NEAT) [9, 5]. These assumptions are implicit: there is, e.g., no explicit proof obligation that states "non-bypassability"; we simply assume that the only security-relevant behavior of a domain is in its kernel API calls.

The NEAT assumptions are well-known, but there are also some less obvious implicit assumptions, that might easily be missed. The fact that the actions are atomic is implicit. We cannot formulate this as a proof obligation in Isabelle. This implicit assumption is, however, crucial for noninterference. This example demonstrates that it is important to make the implicit assumptions explicit and to discuss their validity.

We have split the implicit assumptions behind CISK into four categories: executing actions, interrupts, aborting and waiting. Some implicit assumptions have been marked as *crucial*. If they are not met by the instantiation, then it might be possible that we can formally derive an Isabelle proof of noninterference, even though the system is not secure. The other assumptions are equally important, but do not directly impact soundness.

Executing Actions:

1. **(crucial)** An action is either atomic, *or* any externally visible changes to the state during execution of an action is not security relevant [1].

Alvess-Foss and Taylor identify this implicit assumption for the notion of noninterference of Greve et al. [4, 1] A violation of the security policy can be constructed using some non-atomic action that, e.g., temporarily writes to some illegal part of the memory and then overwrites it correctly before a security check occurs. Such an action could satisfy all explicit proof obligations, and thus the system could be proven secure, even though illegal information flows can exist.

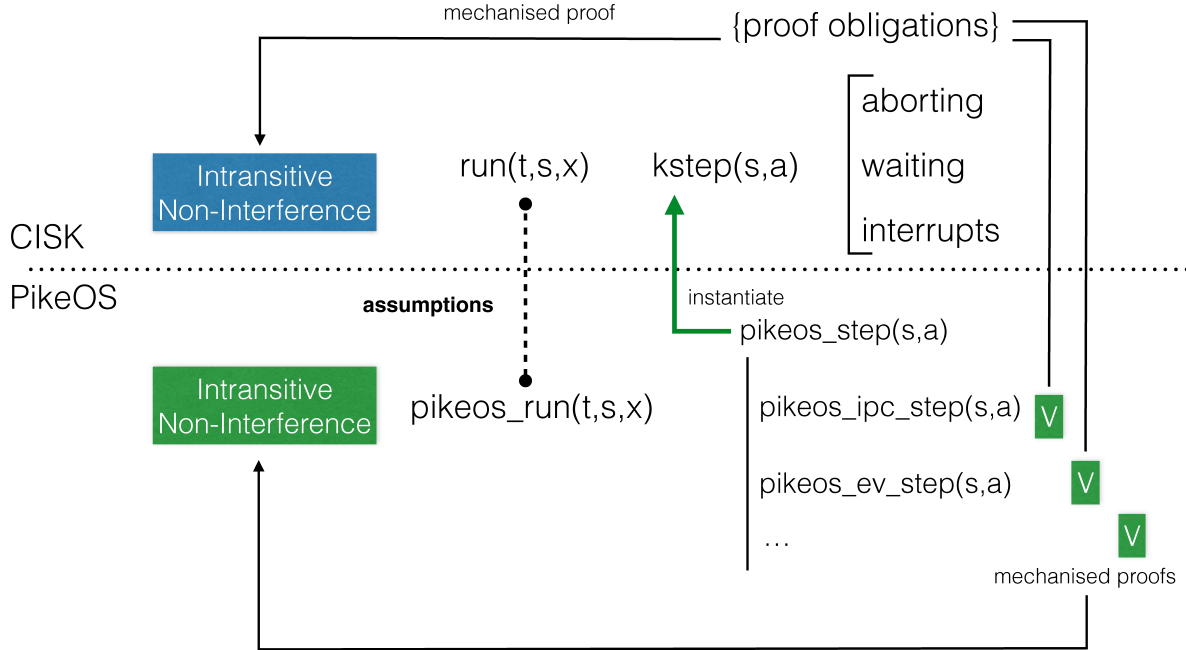


Figure 2: Validating PikeOS against CISK

2. **(crucial)** The kernel runs on single-core hardware. Function *run* models single core behavior. Actions are executed sequentially as time progresses. The step function deterministically computes the next state. Engelhardt et al. show that a definition of noninterference that is sound for a deterministic model, can be unsound in a non-deterministic setting [3]. They show that a definition of noninterference in non-deterministic systems requires taking into account all that could have been observed to the present moment, rather than just the current observation.
3. Actions are independent of the current time. CISK contains an abstract notion of time, used to model interrupts. This notion of time, however, does not reflect a realistic time progression. First, we do not assume time slots to be of equal duration. Instead, each time slot coincides with an atomic action. Second, since we model kernel API calls only, we do not consider time progression when the domains are running in user (non-kernel) mode. As such, CISK does not support the modelling of actions that are time-based.

PikeOS: This is the main reason why we explicitly introduced interrupts in our model: in CISK an IPC call is not atomic. However, once it is split up into three stages, each of these stages is atomic. A timing interrupt will always wait for completion of such a stage. Similarly, all API calls will be split up into atomic stages. A consequence of splitting up API calls into different stages, is that the kernel cannot necessarily execute any given action in any state. This is an implicit assumption underlying existing models [8, 4, 10]. Consider for example the action “IPC_WAIT”. Such

an action can only occur in a state where an “IPC_PREP” has previously been executed (whether or not followed by an interrupt). CISK allows the modelling of these kinds of action sequences. We are working on lifting this assumption, as PikeOS can run on multi-core hardware. For now, certification can only deal with a single-core setting.

Interrupts:

1. Interrupts are state-independent.

The oracle function *interrupt* takes as input the current time and decides whether an interrupt occurs solely on that. Since we assume a single-core setting, any non-trivial instantiation (i.e., with more than one domain) necessarily has some notion of scheduling. This scheduler cannot base its decisions on the current state. It cannot, e.g., decide at some point that because domain *u* is asleep, some other domain *v* can be active. The scheduler must be based on an at beforehand fixed schedule.

Interrupts can stem from an internal scheduler, but also from external devices. The same applies. Whether at some given time a device-interrupt occurs must be independent on the state of the kernel.

2. **(crucial)** An interrupt can change the state of a domain (e.g., modify the memory of that domain), but only in such a way that no information from any other domain is transferred. Essentially, interrupts are read-only. If a device sends an interrupt then it may add information to the state (either the global state or of some domain). However, this information cannot be derived from other domains.

PikeOS: Indeed, the PikeOS kernel has a fixed time schedule. As for device interrupts, most interrupts are attached to one partition (domain) only. As such, they cannot cause any illegal information flow. If they are shared, then we can only prove noninterference when the interrupts are ‘read-only’.

Aborting:

1. If an action is aborted, the domain can still proceed. The control mechanism implements aborting behavior by letting domains whose actions are aborted proceed. This stands in contrast to a model where once an action is aborted the domain enters an error state from which it does not recover.
2. If an action is aborted, it can still produce observable output.

PikeOS: Note that from the kernel point of view, the actions executed by the domains do not matter. If some action is aborted, the kernel notifies the corresponding domain. How this notification is interpreted by the domain, i.e., if it ignores the message or enters an error state, is irrelevant for noninterference. An aborted action results in the transmission of an error code. This error code should be part of the *output* function that computes given the current state and the current action the observable output.

Waiting:

1. If an action is delayed, the entire domain is locked. Per domain, there is one execution. If the current action is delayed, the control mechanism will not select any other action from that execution. Therefore, the delayed action will block the entire domain.

PikeOS: This assumption is valid if we model PikeOS in such a way that there is one thread per domain. PikeOS supports multiple threads per partition (domain), but our way of modelling is more fine-grained and therefore sound.

4. CONCLUSION

Methodologies based on generic models and proof obligations on the step function are attractive, as they reduce the effort of verifying an entire separation kernel to proving properties of methods supported by the API. The drawback of this method lies in the implicit assumptions made in defining the run function from the step function. We presented some of these assumptions and discussed arguments about their validity for PikeOS.

We currently have a detailed model of the IPC and EVENT API calls of PikeOS. We discharged all proof obligations for these actions. We are extending this detailed model to more actions. This will clearly show the feasibility and limits of using CISK and its proof obligations. Within the EURO-MILS project, we have the opportunity to present our arguments about PikeOS and its validity as an instantiation of CISK to Common Criteria evaluators. We expect interesting discussions resulting in a *generic* set of assumptions for which evidence of their validity must be provided. Discharging them for PikeOS should result in a general methodology that should be applicable to a large class of separation kernels.

5. ACKNOWLEDGMENTS

We thank Bruno Langenstein, Werner Stephan (DFKI), Abderrahmane Feliachi, Yakoub Nemouchi, Burkhard Wolff (Univ. Paris-Sud) for fruitful discussions about our model and results. We acknowledge funding from the European Union’s Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 318353 (EURO-MILS project: <http://www.euromils.eu>).

6. REFERENCES

- [1] J. Alves-Foss and C. Taylor. An analysis of the GWV security policy. In *Fifth International Workshop on the ACL2 Prover and its Applications (ACL2-2004)*, 2004.
- [2] J. Brygier, R. Fuchsen, and H. Blasum. PikeOS: Safe and secure virtualization in a separation microkernel. Technical report, SYSGO, 2009.
- [3] K. Engelhardt, R. van der Meyden, and C. Zhang. Intransitive noninterference in nondeterministic systems. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 869–880. ACM, 2012.
- [4] D. Greve, M. Wilding, and W. M. Vanfleet. A separation kernel formal security policy. In *Fourth International Workshop on the ACL2 Prover and Its Applications (ACL2-2003)*, 2003.
- [5] W. S. Harrison, N. Hanebutte, P. Oman, and J. Alves-Foss. The mils architecture for a secure global information grid. *Crosstalk: The Journal of Defense Software Engineering*, 18(10):20–24, 2005.
- [6] R. Kaiser and S. Wagner. Evolution of the PikeOS microkernel. In *First International Workshop on Microkernels for Embedded Systems*, page 50, 2007.
- [7] J. Rushby. Design and verification of secure systems. In *ACM SIGOPS Operating Systems Review*, volume 15, pages 12–21. ACM, 1981.
- [8] J. Rushby. *Noninterference, transitivity, and channel-control security policies*. SRI International, Computer Science Laboratory, 1992.
- [9] G. Uchenick and W. Vanfleet. Multiple independent levels of safety and security: high assurance architecture for msls/mls. In *Military Communications Conference, 2005. MILCOM 2005. IEEE*, volume 1, pages 610–614, Oct 2005.
- [10] R. van der Meyden. What, indeed, is intransitive noninterference? In J. Biskup and J. López, editors, *Computer Security – ESORICS 2007*, volume 4734 of *Lecture Notes in Computer Science*, pages 235–250. 2007.
- [11] F. Verbeek, J. Schmaltz, S. Tverdyshev, H. Blasum, and O. Havle. Intransitive noninterference for separation kernels with interrupts and control. Submitted for publication, 2014.