Electronic Communications of the EASST
Volume 076 (2019)

Automated Verification of Critical Systems 2018
(AVoCS 2018)

Proof-Oriented Design of a Separation Kernel with Minimal Trusted Computing Base

Narjes Jomaa, Paolo Torrini, David Nowak, Gilles Grimaud and Samuel Hym

20 pages

Guest Editors: David Pichardie, Mihaela Sighireanu
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Proof-Oriented Design of a Separation Kernel with Minimal Trusted Computing Base

Narjes Jomaa, Paolo Torrini, David Nowak, Gilles Grimaud and Samuel Hym

CRIStAL, CNRS & University of Lille, France

Abstract: The development of provably secure OS kernels represents a fundamental step in the creation of safe and secure systems. To this aim, we propose the notion of protokernel and an implementation — the Pip protokernel — as a separation kernel whose trusted computing base is reduced to its bare bones, essentially providing separation of tasks in memory, on top of which non-influence can be proved. This proof-oriented design allows us to formally prove separation properties on a concrete executable model very close to its automatically extracted C implementation. Our design is shown to be realistic as it can execute isolated instances of a real-time embedded system that has moreover been modified to isolate its own processes through the Pip services.

Keywords: protokernel, memory isolation, formal proof, Coq

1 Introduction

The development of provably secure kernels addresses the trusted computing base (TCB) that has privileged access to the hardware, thus playing a fundamental role in the development of secure systems [AST06]. Making such development manageable and cost-effective remains a major challenge in formal methods even after major breakthroughs [KEH$^+$09, GSC$^+$16]. Memory management and isolation are at the core of the functionalities of separation kernels [Rus81, ZSZL17], designed to ensure that users can access resources and communicate with each other only according to a given security policy.

The centrality of memory management and access control provides our motivation for introducing the notion of protokernel, as a minimal separation kernel that in fact provides only services related to virtual memory management and context switching between applications. In this paper, we present the development and verification of the Pip protokernel [The17b], relying on a comparatively lightweight approach, yet providing an efficient, real-world application. We use Coq, a proof assistant based on the Calculus of Constructions [BC04], to develop the monadic, executable model of the services on top of a hardware abstraction layer, and we prove security properties based on a low-level memory isolation criterion, using Hoare logic [The17c]. The model is then automatically translated to a small fragment of C, using an Haskell-implemented translator [HO17].

Pip has been developed for industrial application as part of a European project[1]. Its development relies on a proof-oriented design, narrowing the TCB needed to implement the kernel services in a machine independent way on top of the hardware abstraction layer. This design,

[1] This work was funded by the European Celtic-Plus Project ODSI C2014/2-12.

minimalistic yet efficient, is the result of a collaborative effort with our colleagues from the operating system community (cf. [BGI18, YGG+18, BJY+18] for benchmarks and more details) to support on-demand secure isolation [The17a]. Crucially, Pip supports efficient memory management and a dynamic notion of memory separation based on a hierarchical TCB architecture, allowing for isolated partitions to be created and removed after initialisation.

1.1 Contributions

In this paper, we report the formal verification viewpoint of the development of the Pip protokernel. We present: 1) a Coq model that includes the relied-upon hardware components (memory and MMU), an executable specification of the memory manager supporting hierarchical TCB, and a highly modular API consisting of ten services; 2) a low-level specification of memory isolation; 3) an abstract information flow model which allows us to prove a non-influence result; 4) a verification methodology based on a Hoare logic on top of an enhanced state monad, designed to carry out well-structured proofs of memory isolation on the executable model; 5) the application of this methodology to the verification of three of the API services.

1.2 Related work

The development of secure kernels has been a long-standing goal in the formal methods community [ZSZL17]. Memory separation in its basic form can be characterised in terms of access control. Stronger notions of information flow separation have been introduced with the noninterference property [GM82], ensuring that events are not leaked [Rus92], and the non-leakage property [vO04], ensuring that data are not leaked. Early work on access control policies for the UCLA security kernel [WKP80] focused on the verification of an abstract model, stressing the much higher cost of verifying executable code. The notion of separation kernel, originally introduced by Rushby [Rus81] to ensure information flow separation, is at the foundations of security-oriented kernels based on the MILS architecture [AST06] and of safety-oriented ones [ZSZL16].

SeL4 [KEH+09] is a microkernel of the L4 family formally verified in Isabelle-HOL. It provides virtual memory management, thread scheduling, inter-process communication and access control. High-level verification is carried out using Hoare logic on a abstract model. An executable specification that refines the abstract model is obtained by automated translation from a Haskell prototype. The manually written C implementation is proved to refine the executable specification using an automatic translation of C to Isabelle based on the semantics in [TKN07]. The refinement proofs are functional correctness ones, thus quite big, ensuring that the abstract model provides a complete specification of the C behaviour. Security properties are proved relying on the abstract model and functional correctness [MMB+13].

PROSPER is a separation kernel that supports dataflow functionalities of the ARMv7 processor [DGK+13], including inter-partition communication, scheduling, and in an extended version [SD14] direct memory access. Security properties have been proved in HOL4. Bisimulation is used to carry higher-level proofs, based on an abstract model including communication channels, over to a concrete one based on an intermediate language that represents the ARM instruction set [FM10].

CertiKOS [GSC$^+$16] provides a certified programming method to formally develop kernels in Coq, targeting an extension of CompCert Clight and assembly code [BL09]. The specification approach is based on certified abstraction layers, corresponding to Clight external functions, enhanced with a notion of abstract state, to allow for functional correctness specifications in the code. CertiKOS supports fine-grained layering of programming and verification, relying on a notion of contextual refinement [GKR$^+$15], and it has also been used to prove noninterference results [CSG16].

In [BBBT11] the PikeOS memory manager is verified at the C source code level using an automated static verifier, proving that initialisation-time partitioning of memory is preserved at runtime. In [BBCL11] a Xen-based paravirtualization-style hypervisor is abstractly modelled in Coq and information flow separation is proved. In [ZSZL16] Isabelle-HOL is used to model abstractly the ARINC-653 channel-based inter-partition communication for safety-oriented separation kernels. In [BCG$^+$16] an abstract model of the Nova kernel is formalised in Coq, data security properties are proved, and program extraction is used to generate testing code. In [SBH14] an abstract security model is formalised in Isabelle-HOL to verify properties of the XtratuM kernel using refinements.

1.3 Pip

Our system relies on a notion of memory separation, on top of which noninterference can be proved for isolated partitions. Inter-partition communication and scheduling are not part of Pip, and therefore communication policies are not part of the kernel TCB. On the other hand, Pip allows for partition reconfiguration at runtime, and tree-like partitioning can naturally support any user TCB hierarchy. The security proof we provide does not rely on any specific architecture, and has been the driving factor in the development. The verification of memory separation is comparatively lightweight, and this has been achieved by avoiding to go through a full functional correctness proof. From the development point of view, Pip relies on a separation of concerns between modelling and verification on one side and translation to C on the other. Nonetheless, unlike work based on abstract models, Pip provides an executable model of its services, written in a shallow embedding of a small fragment of C. Indeed, our Coq development is based on a direct formalisation of the executable model, relying on consistency properties to capture the abstract requirement in a bottom-up way, following the approach already used in [JNGH18] for a simpler system with a flat memory model. The translation of the service layer to C source code is fully automated, and its verification, on which the correctness of the model does not depend, is under way as an independent workload (see [TNJC18] for more discussion of this point).

1.4 Outline

In Section 2 we present the design of Pip. In Section 3 we present the Coq model, including hardware components and partitioning manager. In Section 4 we present the security properties and the abstract information flow model. In Section 5 we present the isolation proofs and the underlying methodology. Conclusions and further work are discussed in Section 6.

2 Proof-oriented design

Pip is a kind of security kernel conveniently described as a *protokernel* [The17b, BJY$^+$18, BIG17], designed to minimize the TCB. The goal of such minimization is twofold: to reduce the attack surface, and to increase the feasibility of a formal proof. Pip only provides the hardware management that suffices to enforce security in the form of memory separation: in this sense, we speak of proof-oriented design, emphasising the feedback of theorem-proving in the iterative development of the system. Like exokernels [EKO95], Pip does not offer any higher-level abstraction: in particular, it offers neither threads, nor processes, nor a file-system, nor a network stack. Pip provides a runtime environment that is a kind of virtual machine and that we call a *partition*. On top of that we can implement any high-level abstraction. Each partition allows configuring a virtual memory management unit (MMU) and managing virtual interrupt request (IRQ) via dedicated Pip services that form the Pip API. Using these configuration services, it is possible to port on Pip a conventional kernel (e.g. Linux) as well as an embedded system (e.g. FreeRTOS) or even a hypervisor (e.g. Xen). Little effort is needed to adapt one of them to run as a partition (see [BJY$^+$18] for details, including more than satisfactory benchmarks), relying on the paravirtualization approach [BDF$^+$03].

Pip can be better described as protokernel rather than hypervisor because it does not include multiplexing management, thus reducing the kernel TCB to its bare bone. Indeed, in a sense Pip runs a single primary partition. This partition, created at boot time, represents the whole and only real machine. Pip assigns to it the entire address space (except for the memory used by Pip itself) and the whole CPU time (except for the CPU time spent by Pip to handle real IRQs).

Nonetheless, Pip supports a hierarchical partitioning model, allowing each partition to create its own subpartitions and split part of its own address space between them. This recursively results in the creation of a partition tree whose root is the primary one (henceforth the *root partition*). The management of the partition tree takes place according to a recursive scheme. Only one partition can be executing at each time in a monocore system, but Pip does neither multiplex interrupt requests nor share CPU time between distinct partitions. Pip only deals with software interrupts that are system calls to its own API. Other software interrupts are forwarded to the parent of the caller. On the other hand, all the hardware interrupts are forwarded to the root. In this way, Pip delegates multiplexing (i.e. determining the control flow between partitions) to the root, which can implement any possible multiplexing policy, recursively allowing each partition to share CPU time between its children.

Executing multiplexing in user mode allows us to cut drastically the size of the kernel TCB. On the other hand, the hierarchical architecture allows each partition to rely on the environment managed by its parent and to split its environment between its children. This naturally induces a hierarchical characterisation of the TCB associated with each partition, which can be expressed in terms of a simple transitive policy: each partition only relies on its ancestors and the kernel. The primary purpose of this design is to allow for a cost-effective verification that this policy is enforced, by means of a memory isolation proof.

The memory isolation property ensured by Pip is based on hardware functionalities that control physical memory accesses. Partitions can not use physical addresses: in order to access information they can only use virtual addresses that will be converted to physical ones using the Memory Management Unit. Each partition has an MMU configuration, and on every user access, the MMU is invoked to decide on whether authorising or rejecting the access. These decisions are based on the configuration pages, called page tables or indirections, written by the kernel so as to allow translation of any given virtual address to the corresponding physical address. In order to guarantee security, page tables should never be accessible by any entity except the kernel. Pip satisfies this requirement by construction.

Partitioning model

The physical memory of a computer is split into fixed-size chunks called pages which can be allocated, accessed and used to store information. Whenever a partition is created, Pip associates it with a page which we call its partition descriptor (PD), denoted by an identifier referred to as PDI.

The memory is split between primary kernel pages, solely accessible by the kernel, and user pages, initially assigned to the root partition and accessible by it. When a partition creates a child in the course of its execution, it allocates some of its accessible pages exclusively for that child; in particular, the executing partition hands over to the kernel the sole accessibility of some of the newly allocated pages, to be used for the configuration of the newly created partition; the remaining of those pages, already assigned to the parent are also assigned to the child. In line with the hierarchical model, the executing partition can always delete a child (and with it its descendants). When this happens, the parent regains accessibility to the configuration pages of the descendants. Notice that in general, the pages a partition can access are a subset of those assigned to it. We refer to the pages assigned to a partition together with those used for its configuration as the pages allocated for that partition.

The API of Pip is constituted of ten services that can be called by the executing partition, eight of which are for managing its memory space. *createPartition* adds a new child to the caller. *deletePartition* deletes a child and gives all the pages allocated for it back to the caller. *addVaddr* lends physical memory to a child. *removeVaddr* removes a page from a child. *mappedInChild* returns the child to which a given page is assigned. *prepare* gives pages to the kernel to manage a child's configuration. *pageCount* computes the number of pages required to configure a child. *collect* gives pages lent to the kernel back to the calling partition. The remaining two services are for handling IRQs. *dispatch* notifies a partition about a given interrupt. *resume* restores the context of a previously interrupted partition.

Fig. 1 serves to illustrate partition management in Pip: the available memory makes up the root partition $\mathrm{P_{root}}$. User code running in any partition can use the memory management services exposed by Pip to create child partitions, here $\mathrm{P_1}$ and $\mathrm{P_2}$, lending some of its accessible pages to each of them. For instance, code running in $\mathrm{P_1}$ can create $\mathrm{P_{1.1}}$ using available memory in $\mathrm{P_1}$.

While the parent partition can read and write in the memory accessible by its children (we call this property *vertical sharing*), it cannot access anymore to the pages given to Pip (we obviously want to prevent a partition from messing up Pip data structures, we call this property *kernel isolation*). Sibling partitions (i.e. partitions that have the same parent) cannot access each other's memory (we call this property *horizontal isolation*). If siblings want to communicate, they cannot do it through shared memory. Instead, their parent can use the services of Pip to implement a communication protocol, for instance by flipping a memory page between the siblings.

For example, as result of horizontal isolation, the partition $\mathrm{P_1}$ in Figure 1 is disjoint from the

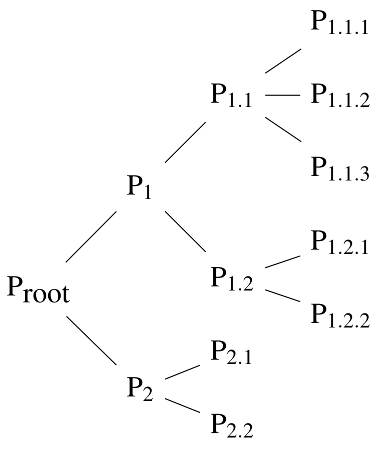

Figure 1: An example of partition tree

partition P_2. Thus, code running in the partition P_1 cannot access the memory in the partition P_2. Moreover, since partitions $P_{1.1}$ and $P_{1.2}$ are included in the partition P_1, they are also disjoint from the partition P_2. However their parent partition P_1 can access their memory and the memory of their descendant partitions $P_{1.1.1}, \ldots, P_{1.2.2}$ as result of vertical sharing.

3 The executable specification

The development of Pip is based on a layered model formalised in Coq, as shown in Fig. 2. At the top level, the service layer provides a source-code level executable specification of architecture-independent system management functions. The service layer consists of the kernel service API that allows for context switching and partition tree management (PTM) which is the core of the Pip engine. The service layer is implemented as an algorithmic model, using the shallow embedding of a C-like fragment based on a monadic encoding, which can be automatically translated to C code. The service layer is built on top of a hardware abstraction layer (HAL), including memory abstraction (MAL) and interrupt abstraction (IAL). The MAL itself can be structured into two layers: hardware memory abstraction (HMAL) and monadic memory abstraction (MMAL). The HMAL provides an abstract model of the physical memory and the MMU. The IAL is also structured into two layers: hardware interrupts abstraction layer (HIAL) and monadic interrupts abstraction layer (MIAL) required to perform context switching. The MMAL and MIAL provide high-level but executable specifications of low-level, architecture-dependent functions which have been manually implemented in C and assembly. The executable model that can be actually run in Coq thus includes the algorithmic model of the service layer as well as the MMAL and MIAL.

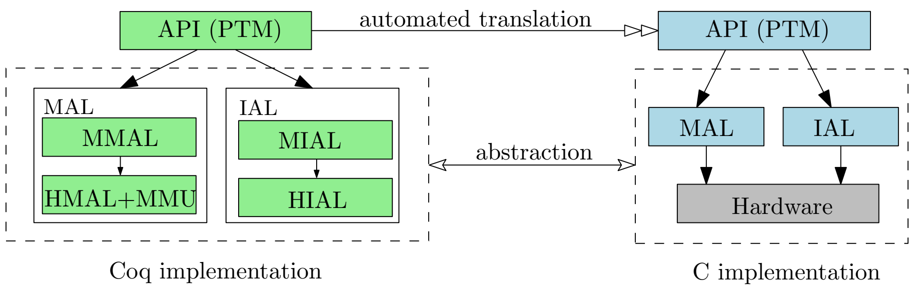

Figure 2: The design of Pip

3.1 Hardware memory abstraction layer

Memory size is determined in our model by two architecture-dependent parameters, the positive integers `pageSize` for memory page size and `nmbPages` for the number of pages. Pages are pointed to by page identifiers of type `page`, which are natural numbers less than `nmbPages`, modelled in Coq using dependent records and defined as `{p :> nat; Hp: p < nmbPages}`. We use 0 as default value for the identifier that points to the null page. The physical address of a memory cell consists of a page identifier and a position in that page, given as an offset value called index. It is modelled by type `paddr` and defined as `(page * index)`. Also indices are typed as a subset of positive integers `index` and defined as `{i:> nat; Hi: i< pageSize}`.

In general, the physical memory state of a computer can be modelled as an association list which maps physical addresses to values. However, our model is completely abstract with respect to the content of the user space. We only need to model the partition tree, and thus we only provide the content of the kernel-owned pages, from which the tree structure can be computed. The part of the hardware state that is relevant to Pip consists of the partition descriptor of the currently executing partition and of the parts of the physical memory where Pip stores its own data (essentially related to the configuration and management of the partition tree), as expressed by the record type `state` which is defined as:

```
Record state: Type:={ currentPartition: page;
                      memory: list(paddr * value)}.
```

Direct memory access is represented by a lookup function named `select`. The `value` datatype sums up the types of values that can be stored by Pip.

```
Inductive value : Type:= |PE: Pentry → value |VE: Ventry → value
  |PP: page → value |VA: vaddr  → value |I:  index  → value.
```

Here `Pentry` stands for physical entry, `Ventry` for virtual entry, and `vaddr` for virtual address. We prove that memory management is well-typed with respect to these value types, as an invariant consistency property of our model. Physical entries (PTEs) make it possible to associate page addresses with information concerning whether each page is assigned to the current partition (`present`) and whether it is owned by it (`accessible`). A physical entry is modelled by

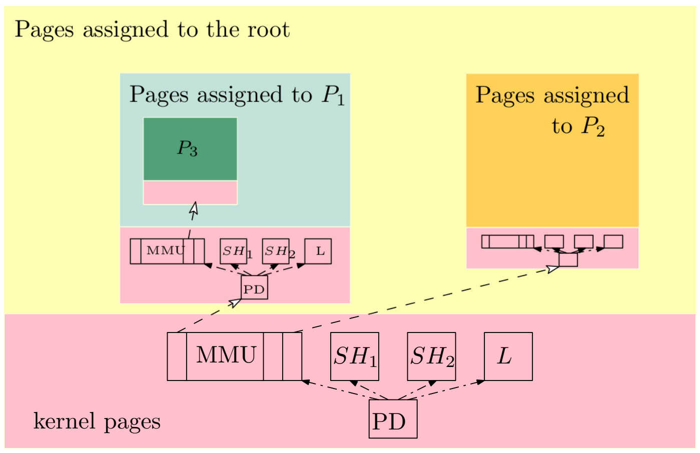

Figure 3: Partition tree configuration

type `Pentry` and defined as `{pa: page; present: bool; accessible: bool}`. A page table is a configuration page that contains a set of values, including physical entries.

3.2 Partition tree management

The configuration of each partition (other than the root) is stored in the memory allocated for it by its parent partition, thus assigned to the parent, though in the part of it that is only accessible to the kernel, and it is defined by four entities. The first and principal one is the MMU configuration, the other ones include two tree-like structures and a list-like one. Each partition descriptor (PD) is a page that contains the addresses of four configuration entities, as illustrated by Fig. 3. The PD together with its entities forms what we call a configuration node.

The MMU configuration of each partition has a hierarchical structure that can be abstractly described as a graph where nodes are pages, each being either a page table or a terminal page, and each terminal page being either a page assigned to the configured partition or the null page. Each entry in a page table points to another page, thus providing the arcs in the graph, while providing accessibility information. The MMU structure has two fundamental properties which have been verified as invariant consistency properties of our model. The first property ensures that the one and only physical entry marked as not present corresponds to the null page – in fact, the MMU configuration only contains pages that are assigned to the parent partition. The second property ensures that the subgraph formed by non-null pages has the structure of a tree, which we call the MMU tree. The tree depth is fixed by the architecture-dependent natural parameter `levelNum`. Virtual addresses are modelled by type `vaddr` as lists of indices of length `levelNum+1` and defined as:

```
Record vaddr := {va :> list index; Hva: length va = levelNum + 1}.
```

Each virtual address is translated either to the null address or to a physical address in a leaf, by interpreting each index in the list as offset in the page table at the corresponding level in the MMU tree. The last index in the list is the offset of the corresponding physical address, which may be accessible or not to the partition, depending on whether the page is owned or not, as specified by the last PTE. Crucially, in Pip only the kernel can access directly physical addresses. Virtual addresses are the objects provided by the MMU to user applications as indirect references to the physical memory. Indirect access relies on the `mmu_translate` function, which models the hardware mapping of virtual addresses to physical ones, and which has been proved to respect the access policy.

Intuitively, non-null virtual addresses correspond to maximal branches in the MMU tree (plus the final offset), and they include those pointing to the descendants in the partition tree. In fact, the content of a virtual address (i.e. the value of the physical address it gets translated to) can be the PDI of another partition (indeed, this is the only kind of content Pip cares for). In this way configuration nodes can be linked together. The resulting structure is a graph which we prove to be a tree, and thus to represent the partition tree, as an invariant consistency property of our model.

The partition tree management makes use of two auxiliary configuration entities, called shadows, that mirror the corresponding MMU tree. The first shadow is used to find out which pages are assigned to children, an information that is needed to ensure horizontal isolation. It uses the type `Ventry` of virtual entries which is defined using the following record:

```
Record Ventry : Type:={pd: bool; va: vaddr}.
```

The flag `pd` indicates if the associated physical address is the PDI of a child partition. The second shadow is used to associate each partition descriptor to the virtual address it has in its parent partition. The fourth auxiliary entity is a linked list used to remember which pages have been lent to the kernel. The information in the second shadow and in the linked list is needed to process efficiently the deletion of a partition.

3.3 The monad

In order to implement the Pip services imperatively in Coq as sequential programs, we rely on a monadic approach. Monads [Mog91, Wad92] allow threading effects through computations, and can be used to interpret semantically imperative languages in purely functional ones. In our case, side effects include the state, corresponding to the hardware state, and a notion of exception corresponding to the possibility of undefined behaviours that we actually prove to never happen. For example, any attempt by Pip to access a physical address that is not defined in the memory state would result in an undefined behaviour.

We define our monad (Low Level Interface monad) as an abstract datatype `LLI A` that wraps together hardware state and undefined behaviours, following [Mog91, Wad92].

```
Definition LLI (A : Type) : Type := state → result (A * state).
```

Here `result` is an inductive type with two constructors: the first one corresponds to a result of type `A`, and the second one to an undefined behaviour.

```
Inductive result (A : Type) : Type :=
  |val: A → result A |undef: nat → state → result A.
```

We define the monadic operations `ret` and `bind`, which can be easily proved to satisfy the monadic laws [Mog91, Wad92].

```
Definition ret : A → LLI A := fun a s ⇒ val (a, s).
Definition bind : LLI A → (A → LLI B) → LLI B :=
 fun m f s ⇒ match m s with
 |val (a, s') ⇒ f a s' |undef a s' ⇒ undef a s' end.
```

We write `perform x := m in e` for the monadic binding operation `bind m (fun x ⇒ e)`, which can also be written `m ;; e` when `x` is not used in `e`, to represent sequential composition of the stateful actions `m` and `e`. The state monad ensures we can define the usual stateful functions, e.g. `get` and `put` to read and update the state. We use the monadic functions to define the specific ones that form the MAL, matching the corresponding architecture-dependent implementations in C and assembly. Crucially, the Pip services are implemented so to access the state only through the MMAL functions.

3.4 Monadic memory abstraction layer

The MMAL specifies a C and assembly library for the architecture-dependent part of Pip, which consists of kernel atomic operations, such as reading and writing values in physical memory, and performing simple computations and comparisons on them. This includes physical memory access operations needed to configure partitions, virtual memory activation and a set of auxiliary operations. It mainly consists of bitwise operations of constant computational complexity.
The functions defined in the MMAL can be organized along four categories. The first one is `HAL_write`, allowing the kernel to write values into physical memory through physical addresses and including the primitives `writeVirtual`, `writePhysical`, `writeVirEntry`, `writePhyEntry`, `writeAccessible`, `writePresent`, `writePDflag`, `writeIndex`. For example, the `writeVirtual` primitive has the following definition:

```
Definition writeVirtual (addr:page)(idx:index)(va:vaddr): LLI unit:=
 modify (fun s ⇒ {| currentPartition := s.(currentPartition); memory
 := add paddr idx (VA va) s.(memory) beqPage
 beqIndex|}).
```

It stores a virtual address at the physical address given by a page and an index. The monadic operator `modify` is used to modify the association list that represents the memory.
The second category is `HAL_read`. It is about reading values from physical memory and includes the following primitives: `readVirtual`, `readPhysical`, `readVirEntry`, `readPhyEntry`, `readAccessible`, `readPresent`, `readPDflag`, `readIndex`. For example, `readVirtual` allows reading a virtual address from a physical location.

```
Definition readVirtual (p: page) (idx: index): LLI vaddr:=
perform s := get in
```

```
match lookup p idx s.(memory) beqPage beqIndex with
 |Some (VA v) ⇒ ret v |Some _ ⇒ undefined 3 |None ⇒ undefined 2
end.
```

This requires that there is a value of the right type at the given physical address. If the condition is not met, the result is an undefined behaviour.
The third category (named `HAL_op`) allows the kernel to perform simple operations, including e.g. successor and equality comparison, on each value type. The last category (named `HAL_const`) is about accessing global constants. Notice that since Coq types are more discriminating than C ones, different MMAL functions may be actually implemented as the same C function.

3.5 Linguistic aspects

The program extraction capabilities of Coq made it possible to extract Haskell code which we used to test our model. However, such code does not meet the runtime requirements of a realistic kernel. The cost of the runtime environment needed by functional programs, and particularly garbage collection, stand in the way of a reliable runtime behaviour. Therefore, we rely on a translation to C. The monadic code we use in our executable specification corresponds to a quite simple imperative language: essentially, a first-order sequential language with call-by-value, primitive recursion and mutable references. From a low-level point of view, all the datastructures used by Pip, including trees, can be treated as linked lists. The abstract memory model in Coq matches closely this low-level characterisation: Pip datastructures are represented using an encoding of linked lists, given in terms of lists and access to the monadic state, rather than by using specific inductive datatypes, and then by proving invariant consistency properties as appropriate to ensure that the low-level representation is correct.

In fact, our monadic code corresponds to a shallow embedding of the denotational semantics of a language, formally described in [TNJC18], which matches closely a comparatively small fragment of C (we use no arrays, neither structures nor unions, no loop instructions, no pointer arithmetics). We use a Haskell-implemented tool called *Digger* [HO17] to generate automatically C code for the service layer from the abstract syntax of the monadic Coq code. The code generated by *Digger* from the executable model can then be compiled (we currently use GCC) together with the manually implemented MAL functions.

Relying on a low-level modelling of data structures has a drawback in the development of recursive functions based on such structures: we cannot rely on structural recursion as Coq would provide for the corresponding inductive type. This can be a problem already at the level of function definition, as Coq requires that we ensure termination. Nonetheless, we can easily deal with this issue, relying on the fact that Pip functions only require recursion bounded by parameters of the hardware architecture such as the size of a memory page. However, this means we need to prove that fuel suffices, for each top-level function call.

4 Security properties

Security in Pip relies on the MMU functionalities that control user access to physical memory, as user access to physical memory is only allowed through the MMU. Proved consistency properties ensure, for instance, that the MMU is configured correctly, consistently with the partition tree, and that memory access through the MMU is consistent with the access policy encoded in the configuration pages. Such properties are needed to prove the actual security properties of Pip. We now introduce the main properties that define the Pip partitioning model, provably preserved by the API services: horizontal isolation, kernel isolation, and vertical sharing.

In a tree, two nodes are unrelated whenever neither of them is a descendant of the other one. With respect to the partition tree, horizontal isolation means that unrelated partitions cannot access each other's memory. This is the case in particular for sibling partitions.

Definition 1 (HI) *Horizontal isolation* holds for a state s whenever for all partition descriptors $m_0, m_1, m_2 \in \mathsf{partition\_tree}(s)$, with $m_1, m_2 \in \mathsf{children}(m_0, s)$ and $\mathsf{pdi}(m_1) \neq \mathsf{pdi}(m_2)$, it holds that
$\mathsf{allocated\_pages}(m_1, s) \cap \mathsf{allocated\_pages}(m_2, s) = \emptyset$.

where partition_tree is the list of partition identifiers. We use consistency properties to ensure that partitions are organized into a tree-like structure.

Kernel isolation means that no partition can access the pages owned by the kernel.

Definition 2 (KI) *Kernel isolation* holds for a state s whenever for each partition descriptor $m \in \mathsf{partition\_tree}(s)$, it holds that
$\mathsf{accessible\_pages}(m, s) \cap \mathsf{kernel\_owned\_pages}(s) = \emptyset$.

Vertical sharing means that all the pages allocated for a partition are included in the pages assigned to its ancestors, and therefore, owing to the access policy implemented by the MMU, a partition has read and write access on all the memory accessible by its descendants.

Definition 3 (VS) *Vertical sharing* holds for a state s whenever for all partitions $m_0, m_1 \in \mathsf{partition\_tree}(s)$, with $m_1 \in \mathsf{children}(m_0, s)$, it holds that
$\mathsf{allocated\_pages}(m_1, s) \subset \mathsf{assigned\_pages}(m_0, s)$.

Relying on these invariants we can prove formally a non-influence property for *isolated* partitions [The17d] (i.e. excluding inter-partition communication), relying on an abstract information flow model. We call *P-machine* a state machine at the level of physical memory, where transition steps are specified by the type $\mathsf{pstep}\ (ls : \mathsf{list\ page})\ :\ \mathsf{state} \rightarrow \mathsf{state}$. Here ls is the list of pages which can be accessed in the execution of the physical action associated with the step. We can define state equivalence with respect to a list of pages (where select returns the value stored at a location).

$$s_1 \sim^{ps} s_2 \ := \ \forall (p : \ \mathsf{page})\ (i : \ \mathsf{index}),\ p \in ps \rightarrow \mathsf{select}\ s_1\ p\ i \ = \ \mathsf{select}\ s_2\ p\ i$$

We write $s_1 \sim s_2$ for $s_1 \sim^{ps} s_2$ when ps are all the pages, and $s_1 \sim^{p} s_2$ for $s_1 \sim^{[p]} s_2$. We can specify the access requirement intended by pstep in terms of two conditions. First, the *read* *condition* states that the step depends only on the values read at locations in *ps* (i.e. the physical action depends only on its accessible locations).

$$\begin{array}{l}\mathsf{readCond}\ (\pi : \mathsf{pstep})\ :\ \mathsf{Type}\ := \\ \quad \forall (ps :\ \mathsf{list\ page})\ (s_1\ s_2 :\ \mathsf{state}),\ s_1 \sim^{ps} s_2\ \rightarrow\ \pi\ ps\ s_1\ \sim^{ps}\ \pi\ ps\ s_2\end{array}$$

Second, the *write condition* states that the only locations possibly affected by the step are those in *ps* (i.e. the physical action only affects its accessible locations).

$$\begin{array}{l}\mathsf{writeCond}\ (\pi : \mathsf{pstep})\ :\ \mathsf{Type}\ := \\ \quad \forall (ps :\ \mathsf{list\ page})\ (s :\ \mathsf{state})\ (p :\ \mathsf{page}),\ \neg\, p \in ps\ \rightarrow\ s\ \sim^{p}\ \pi\ ps\ s\end{array}$$

Any state machine can be modelled as a P-machine. Runs can be modelled as action sequences. We can now introduce *V-machines* as virtual-level state machines which lift our information flow specification at the level of virtual memory. We can define the notion of virtual step by lifting a physical step, relying on the MMU translation function that represents virtual memory access in the HAL. Here m is the executing partition and vs is the list of virtual addresses accessed in the execution of the virtual action associated with the step.

$$\begin{array}{l}\mathsf{vstep}\ (p :\ \mathsf{page})\ (\pi :\ \mathsf{pstep})\ (vs :\ \mathsf{list\ vaddr})\ (s :\ \mathsf{state})\ :\ \mathsf{state}\ := \\ \quad \mathsf{let}\ ps\ :=\ \mathsf{map}\ (\mathsf{mmu\_translate}\ p)\ vs\ \mathsf{in}\ \pi\ ps\ s\end{array}$$

The fact that the MMU is configured to return physical addresses in the memory owned by the executing partition, together with the access requirement on the P-machine steps, makes it straightforward in Coq to prove non-influence (a notion defined in [vO04]), with respect to a policy that excludes communication between separated partitions.

We write $@m$ to denote all the pages that are in a partition with address m. We write $s_1 \sim^{m!ps} s_2$ whenever $s_1 \sim^{ps} s_2$, the executing PDI is m in both states, and $s_1 \sim^{@m} s_2$. We prove stepwise non-leakage, defined following [vO04].

$$s_1 \sim^{a!as} s_2\ \rightarrow\ \forall pt\ vs,\ \mathsf{vstep}\ a\ pt\ vs\ s_1 \sim^{as} \mathsf{vstep}\ a\ pt\ vs\ s_2$$

This property can be easily extended to action sequences. Similarly we can prove a noninterference-related property (called local-respect in [Rus92]).

$$\neg\, a \in @a_x\ \rightarrow\ \forall pt\ vs,\ \mathsf{vstep}\ a_x\ pt\ vs\ s \sim^{a} s$$

This, together with non-leakage suffices to prove classical noninterference and non-influence [Rus92, vO04]. Our proof shows how the low-level invariants (HI, KI and VS) can be used to prove a higher-level result. The extent of the result presented here is comparatively limited, as we consider a definition of virtual step which is very strict, ruling out the mechanism we need for inter-partition communication (i.e. interrupts). On the other hand, this model could be extended to deal with given user-level communication policies, by a reformulation of the write condition that takes interrupts into account.

5 The verification approach

5.1 Hoare logic on top of a state monad

Our verification targets the properties KI, HI and VS, defined in Section 4, as well as the consistency properties (which we will denote jointly by C in the following), with the goal of ensuring that each system call to an API service preserves all of them. Thus our proofs consist mainly of reasoning about invariants. We do this by using a Hoare logic which we define on top of the LLI monad. We use the syntax $\{\{\ P\ \}\}\ m\ \{\{\ Q\ \}\}$ for our Hoare triples, defined as follows.

```
Definition hoareTriple {A: Type} (P: state → Prop) (m: LLI A)
(Q: A → state → Prop): Prop :=
 ∀ s, P s → match m s with
             |val (a,s') ⇒ Q a s' |undef _ _⇒ False
            end.
```

Here the unary predicate P gives the precondition and the binary predicate Q the postcondition of running the computation m. If performing m yields an undefined behavior, the triple does not hold, thus ensuring a basic functional correctness property: the services of Pip never lead to an undefined behavior. We created a library of general Hoare logic rules to reason about monadic programs. We also created a specific library of Hoare triples (in general, either weakest preconditions or strongest postconditions) for the MAL primitives that sequentially form each of the API service. For example, the following has the weakest precondition of readVirtual as precondition.

```
Lemma readVirtualWP  table idx  (P : vaddr → state → Prop) :
{{fun s⇒ exists e: vaddr,
lookup table idx s.(memory) beqPage beqIndex = Some (VA e) ∧
P e s}} MAL.readVirtual table idx {{P}}.
```

Our use of Hoare logic relies on the assumption that each system call is atomic: it is required that interrupts are blocked during a system call. This is one of the reasons the system calls were designed to be elementary. Moreover, our Hoare logic deals only with the global state, therefore it is required that in multicore architectures at most one core is executing a system call at any time. Concerning the hardware, we naturally assume that the physical memory behaves correctly, as specified (i.e. essentially it is not volatile), and that so does the MMU component. We also assume that the non-algorithmic MMAL is correctly implemented in C and assembly, and that the system was booted in an isolated and consistent state (as our proof is essentially a preservation one).

The code of Pip has a sequential character. Thus, verification can proceed backward, relying on weakest precondition triples, or forward relying on strongest postcondition ones. The two approaches are actually equivalent in our case, since we can ensure termination and our triples rule out undefined behaviours. In our concrete proofs we found it more convenient to move forward, following the actual execution flow.

5.2 The separation proof for addVaddr

In the case of AddVaddr, the top level invariant is the following:

```
Lemma addVaddrInvariant (src dst child: vaddr):
{{fun s ⇒ HI s ∧ KI s ∧ VS s ∧ C s}} addVaddr src dst child
{{fun _ s ⇒ HI s ∧ KI s ∧ VS s ∧ C s }}.
```

The AddVaddr service is called by the executing partition p_0 to assign one of its pages (further down denoted by srcp) to one of its children (child). Here src is the virtual address of srcp in the address space of p_0, whereas dst is the virtual address that gets associated with srcp in the address space of child. The first part of the service consists in making some checks, before making any update to the state. These checks include consistency ones needed to avoid undefined behaviours, and security ones on each of the given parameters. If any of the security checks fails, the service will abort leaving the state unchanged. To ensure that child is actually a child of p_0, AddVaddr starts by checking if the PDI of child is included in the first shadow of p_0. Then it makes sure that srcp is already assigned to p_0 and accessible by it, checking the control bits present and accessible of the corresponding table entry in the MMU tree of p_0. It checks that srcp is not already assigned to any child using the flag `pd` stored in the first shadow of p_0. Finally it verifies that dst is not already associated to a physical page by going through the MMU tree of child. This prevents overwriting data. These properties gets propagated through the sequential execution until the last part, where state updates take place. In fact, it is part of our design of the Pip services to carry out state updates in a way that maximises invariant propagation without compromising efficiency. In the case of AddVaddr state updates are confined to the last three instructions, for which the following triple holds:

```
{{fun s ⇒ HI s ∧ KI s ∧ VS s ∧ C s ∧ isChild child s ∧
           isPresent srcp s ∧ isAccessible srcp s ∧
           notShared srcp s ∧ isEmpty dst s}}
writeVirtual shadow2TableDst dst src;;
writeVirEntry shodow1TableSrc dst child;;
writePhyEntry MMUtableDst dst srcp
{{fun _ s ⇒ HI s ∧ KI s ∧ VS s ∧ C s}}.
```

writeVirtual modifies the second shadow of child, by storing src at dst (thus ensuring that the page can be efficiently located in the parent address space when it needs to get revoked). writeVirEntry starts the process of assigning the page associated with src to the partition associated with child, by storing the address child at src in the first shadow of p_0. writePhyEntry concludes the process, by associating the physical page srcp to dst in the MMU tree of child. As a result of relying on comparatively flat representations (e.g. our trees are represented using lists), a significant part of the proof consists of ensuring that consistency properties are preserved through sequential updates. This is particularly the case for the consistency of the first shadow, a data structure which has been specifically put in place to support isolation, by ensuring that a page cannot be assigned to distinct children. Our policy of ordering steps to maximise propagation pays with respect to several properties: for example accessibleChildPageIsAccessibleIntoParent, requiring that each page assigned to a partition has a back-pointer in its second shadow, is preserved by the fact of performing writeVirtual first. However, there are some propagated properties that get temporarily broken and need to be patched up with weaker ones. For example, the property notShared srcp s, stating that srcp is not marked as shared according to the kernel information stored at p_0, is no more valid after executing writeVirEntry. Luckily, a weaker property that here we denote H holds. It states that srcp is not assigned to any child according to MMU configuration, and suffices to prove the final triple:

```
{{fun s ⇒ HI s∧ KI s ∧ VS s ∧ C s ∧ isChild child s ∧
           isPresent srcp s ∧ isAccessible srcp s ∧
           H srcp s ∧ isEmpty dst s}}
writePhyEntry MMUtableDst idx srcp
{{fun _ s ⇒ HI s ∧ KI s ∧ VS s ∧ C s}}.
```

In particular, proving HI s (as by definition 1) involves showing that srcp is assigned to child and not assigned to any distinct sibling partition. Proving KI s (as by definition 2) involves showing that srcp is not a kernel page. Finally proving VS s (as by definition 3) involves showing that srcp is assigned to the parent partition of child.

5.3 Overview of the development and the verification

The top level executable specification includes about 1300 lines of Gallina. We estimate 3 months for kernel conception and 9 months for its development. We have currently verified three services: *createPartition*, *addVaddr* and *mappedInChild*. We started from the proof of *createPartition* which introduced the majority of the consistency properties resulting in about 60K lines of proof as detailed in table 1. The other two services were proved relying largely on lemmas already proved for *createPartition* and resulting in about 20K additional lines of proof. The current proof has required about one person-year of verification time. Along the proof, we made several changes to the code of the kernel. Most of time it was about fixing bugs such as adding some missing checks or reordering instructions to simplify the proof.

Invariants	**Lines of proof**	**duration**
createPartition ($\approx 300 loc$)	≈ 60000	$\approx$ 10 months
createPartition + addVaddr ($\approx 110 loc$)	≈ 78000	$\approx$ 2 monthes
createPartition + addVaddr + mappedInChild ($\approx 40 loc$)	≈ 78300	$\approx$ 4 hours

Table 1: The overview of verification

6 Conclusions and further work

In this paper we have presented the formalization of the protokernel Pip which is written in Coq and automatically translated to C code. It supports a hierarchical partitioning model that only requires managing partition virtual spaces and context switching. The main security property ensured by the kernel is memory separation. Pip was designed with special attention to increasing the feasibility of its verification. In this sense, not only the minimisation of the TCB, but also the modular design of the API, the structuring of the configuration information, and the modelling economy of the monadic encoding have played a role in allowing for comparatively simple proofs.

Applications

In order to show that our proof-oriented design is realistic, the real-time embedded system FreeRTOS and Linux 4.10.4 have been ported to Pip [YGG+18, BJY+18]. The porting consists essentially in removing privileged instructions and other operations, replacing them with system calls to Pip, thus allowing for isolating tasks by running them in isolated partitions. Pip has also been extended to support different multi-core configurations [BGI18], and this has been possible without changing its current implementation. The modification of existing proofs on the mono-core version to account this extension is still under study.

Translations to C and DEC

A verified translation to C of the service layer is currently being implemented, following one of the work plans discussed in [TNJC18], relying on a small intermediate language called DEC [TN17] which has been formalised as a deep embedding to allow for a definition in Coq of a translation to CompCert C [BL09]. The *Digger* translator [HO17], which works on the AST of the shallow embedding and is currently used to obtain uncertified C code, can also be used to translate automatically the service layer to DEC. Ongoing work is focusing on proving the adequacy of the deep embedding with respect to the shallow one. In [TNJC18] the deep embedding has also been used to experiment with syntax-driven semi-automation of Hoare logic proofs.

Ongoing and future work

Concerning the formal track, the verification of the remaining services is currently under way, and so is the verification of the translation to C. Future work could involve, on one hand, extending the information flow model to deal with user-level interpartition communication, and on the other hand, extending verification to the architecture-dependent part.

Bibliography

[AST06] J. Alves-Foss, W. Scott Harrison, C. Taylor. The MILS architecture for high-assurance embedded systems. *International journal of embedded systems*, pp. 239–247, 2006.

[BBBT11] C. Baumann, T. Bormer, H. Blasum, S. Tverdyshev. Proving memory separation in a microkernel by code level verification. In *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2011 14th IEEE International Symposium on*. Pp. 25–32. 2011.

[BBCL11] G. Barthe, G. Betarte, J. D. Campo, C. Luna. Formally verifying isolation and availability in an idealized model of virtualization. *FM*, pp. 231–245, 2011.

[BC04] Y. Bertot, P. Casteran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[BCG$^+$16] H. Becker, J. M. Crespo, J. Galowicz, U. Hensel, Y. Hirai, K. Nakata, J. L. Sacchini, H. Tews, T. Tuerk. Combining mechanised proofs and model-based testing in the formal analysis of a hypervisor. In *FM 2016: Formal Methods: 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings 21*. Pp. 69–84. 2016.

[BDF$^+$03] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. H., R. Neugebauer, I. Pratt, A. Warfield. Xen and the Art of Virtualization. 2003.
doi:10.1145/1165389.945462

[BGI18] Q. Bergougnoux, G. Grimaud, J. Iguchi-Cartigny. Porting the Pip proto-kernel's model to multi-core environments. In *IEEE-DASC'18*. P. 8 pages. 2018.

[BIG17] Q. Bergougnoux, J. Iguchi-Cartigny, G. Grimaud. Pip, un proto-noyau fait pour renforcer la sécurité dans les objets connectés. In *Conférence d'informatique en Parallélisme, Architecture et Système (ComPAS), Jun 2017, Sophia Antipolis, France*. 2017. 8 pages.

[BJY$^+$18] Q. Bergougnoux, N. Jomaa, M. Yaker, J. Cartigny, G. Grimaud, S. Hym, D. Nowak. Proved Memory Isolation in Real-Time Embedded Systems through Virtualization. 2018. Draft.

[BL09] S. Blazy, X. Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, pp. 263–288, 2009.

[CSG16] D. Costanzo, Z. Shao, R. Gu. End-to-end verification of information-flow security for C and assembly programs. *ACM*, pp. 648–664, 2016.

[DGK$^+$13] M. Dam, R. Guanciale, N. Khakpour, H. Nemati, O. Schwarz. Formal verification of information flow security for a simple ARM-based separation kernel. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. CCS '13, pp. 223–234. ACM, 2013.
doi:10.1145/2508859.2516702

[EKO95] D. R. Engler, M. F. Kaashoek, J. O'Toole, Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. *SIGOPS Oper. Syst. Rev.*, 1995.

[FM10] A. C. J. Fox, M. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set. *Interactive Theorem Proving*, pp. 243–258, 2010.

[GKR$^+$15] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. N. Wu, S. C. Weng, H. Zhang, Y. Guo. Deep specifications and certified abstraction layers. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '15, pp. 595–608. ACM, 2015. doi:10.1145/2676726.2676975

[GM82] J. A. Goguen, J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*. Pp. 11–11. 1982.

[GSC$^+$16] R. Gu, Z. Shao, H. Chen, W. S. C., J. Kim, V. Sjoberg, D. Costanzo. CertiKOS: an extensible architecture for building certified concurrent OS kernels. In *OSDI*. Pp. 653–669. 2016.

[HO17] S. Hym, V. Oudjail. Digger. 2017. https://github.com/2xs/digger.

[JNGH18] N. Jomaa, D. Nowak, G. Grimaud, S. Hym. Formal proof of dynamic memory isolation based on MMU. *Sci. Comput. Program.* 162:76–92, 2018.

[KEH$^+$09] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tich, S. Winwood. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. Pp. 207–220. 2009.

[MMB$^+$13] T. Murray, D. Matichuk, M. Brassil, T. Bourke, S. Seefried, C. Lewis, X. Gao, G. Klein. seL4: from general purpose to a proof of information flow enforcement. In *IEEE Symposium on Security and Privacy (SP)*. Pp. 415–429. 2013.

[Mog91] E. Moggi. Notions of Computation and Monads. *Information and computation*, pp. 55–92, 1991.

[ODS17] ODSI Partners. http://celticplus-odsi.org. 2017.

[vO04] D. von Oheimb. Information Flow Control Revisited: Noninfluence = Noninterference + Nonleakage. In *Computer Security - ESORICS 2004, 9th European Symposium on Research Computer Security, Sophia Antipolis, France, September 13-15, 2004, Proceedings*. Pp. 225–243. 2004.

[Pip17a] Pip Development Team. http://pip.univ-lille1.fr. 2017.

[Pip17b] Pip Development Team. http://pip.univ-lille1.fr/doc/coq-doc/toc.html. 2017.

[Pip17c] Pip Development Team. https://github.com/2xs/pipcore/tree/develop. 2017.

[Rus81] J. Rushby. *Design and verification of secure systems*. ACM, 1981.

[Rus92] J. Rushby. *Noninterference, transitiviy, and channel-control security policies*. SRI International, Computer Science Laboratory, 1992.

[SBH14] D. Sanan, A. Butterfield, M. Hinchey. Separation kernel verification: the XtratuM case study. In *Working Conference on Verified Software: Theories, Tools, and Experiments*. Pp. 133–149. 2014.

[SD14] O. Schwarz, M. Dam. Formal verification of secure user mode device execution with DMA. In *Haifa Verification Conference*. Pp. 236–251. 2014.

[TKN07] H. Tuch, G. Klein, M. Norrish. Types, Bytes, and Separation Logic. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '07, pp. 97–108. ACM, 2007. doi:10.1145/1190216.1190234

[TN17] P. Torrini, D. Nowak. https://github.com/2xs/dec.git. 2017.

[TNJC18] P. Torrini, D. Nowak, N. Jomaa, M. S. Cherif. Formalising executable specifications of low-level systems. In *Proc. VSTTE'18*. P. 18 pages. 2018.

[Wad92] P. Wadler. Comprehending Monads. *Mathematical Structures in Computer Science*, pp. 461–493, 1992.

[WKP80] B. J. Walker, R. A. Kemmerer, G. J. Popek. Specification and verification of the UCLA Unix security kernel. *Communications of the ACM*, pp. 118–131, 1980.

[YGG$^+$18] M. Yaker, C. Gaber, G. Grimaud, J.-P. Wary, V. Sanchez-Leighton, I.-C. J., X. Han. Ensuring IoT security with an architecture based on a separation kernel. In *Fi-Cloud'18*. P. 8 pages. 2018.

[ZSZL16] Y. Zhao, D. Sanan, F. Zhang, Y. Liu. Reasoning about information flow security of separation kernels with channel-based communication. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Pp. 791–810. 2016.

[ZSZL17] Y. Zhao, D. Sanan, F. Zhang, Y. Liu. High-assurance separation kernels: a survey on formal methods. *arXiv preprint arXiv:1701.01535*, 2017.