On Demand Secure Isolation

D1.1 – Abstract Model of the Isolation Manager (public)

Project acronym	ODSI
Project title	On Demand Secure Isolation
Advantage	
Deliverable number	D1.1
Deliverable name	Abstract Model of the Isolation Manager
Version	V 1.0
Work package	WP 1 – Minimal Isolation
Lead beneficiary	LILLE
Authors	N. Jomaa, G. Grimaud, S. Hym, D. Nowak, P. Torrini (Lille)
Nature	R – Report
Dissemination level	PU – Public
Delivery date	26/06/2017 (M20)

History

Version	Date	Description
0.1	16 March 2017	Preliminary draft
1.0	26 June 2017	Final version

Executive Summary

The ODSI platform is based on proved kernels to ensure strong security properties compatible with higher level of certifications. Two models of kernels are developed for two different architectures:

- The ODSI Hypervisor for ARM / Trustzone architecture
- The ODSI Mesovisor for Intel architecture

The ODSI Mesovisor (called Pip in the rest of the document) is a protokernel: it allows for kernels, ranging from hypervisors to monolithic kernels, to be developped as user mode applications. This means that only the code of Pip is executed in kernel mode (i.e., the privileged mode of the hardware). Indeed, code running in kernel mode has direct access to the whole memory and hardware. It is thus clearly better, from a security point of view, to keep this code as minimal as possible. This stems from the general principle that the trusted computing base (TCB) should be kept minimal.

Pip only provides system calls for the management of partitions of the memory and for the basic dealing of control flow, thus reducing the trusted computing base to its bare minimum. More precisely, Pip provides 9 services that can be called from any partition. Only two of them are for basic control flow, the remaining ones are for memory management. For instance, Pip provides services to create or delete child partitions.

As the main contribution of this deliverable, we show how we were able to model the code of the Pip kernel in Gallina — the language of Coq proof assistant — in an imperative style that allows for direct translation into freestanding C. In order to ensure semantic soundness in the translation from the executable model of Pip, written in Gallina, to its implementation in C, we rely on a reflective approach. We have developed a domain specific language in Coq, designed as the target for a semantic interpretation of the relevant sub-language of C. We are implementing an interpreter for this language in Gallina, that will be reversed to provide an operational specification of Pip. The domain specific language will then be used to translate Pip directly to the CompCert C front-end.

Pip relies on a thin layer of low-level code, the memory abstraction layer (MAL), that gives Pip access to the physical memory. Another of our contributions is an executable model of this MAL that summarizes what the developers of Pip assume about it. It allows for making proofs but also for simulating inside Coq the execution of Pip.

Our last contribution is a formal, yet readable, formalization of the security features provided by Pip to applications, essentially memory isolation, and consistency properties that are discovered while making the proof of isolation.

Table of contents

List of figures

1 Oversiew

1.1 Introduction

The operating system (OS) is one of the most crucial part of a computer system. It allows untrusted applications to share computer resources. Thus, for both safety and security reasons, it is important to prevent accidental and malevolent access by a process to an address outside its own address space. That is to say that the OS must ensures a high level of security in order to protect data from malicious accesses.

The Trusted Computing Base (TCB) is that part of the system that we can trust to ensure the security of the whole system. Given the complexity of the OS, the TCB should be kept as small as possible and only include functions which are critical for security. The TCB typically includes of the kernel of the OS. Memory isolation between applications is guaranteed by the kernel which implements it with the help of a piece of hardware called memory management unit (MMU) through which all memory accesses must go.

In this deliverable, we focus on the protokernel Pip[1]. It allows for kernels, ranging from hypervisors to monolithic kernels, to be developped as user mode applications. This means that only the code of Pip is executed in kernel mode (i.e., the privileged mode of the hardware). Indeed, code running in kernel mode has direct access to the whole memory and hardware. It is thus clearly better, from a security point of view, to keep this code as minimal as possible. This stems from the general principle that the TCB should be kept minimal.

Contributions. As the main contribution of this deliverable, we show how we were able to model the code of the Pip kernel in Gallina — the language of Coq proof assistant — in an imperative style that allows for direct translation into freestanding C. In order to ensure semantic soundness in the translation from the executable model of Pip, written in Gallina, to its implementation in C, we rely on a reflective approach. We have developed a domain specific language in Coq, designed as the target for a semantic interpretation of the relevant sub-language of C. We are implementing an interpreter for this language in Gallina, that will be reversed to provide an operational specification of Pip. The domain specific language will then be used to translate Pip directly to the CompCert C front-end.

Pip relies on a thin layer of low-level code, the memory abstraction layer (MAL), that gives Pip access to the physical memory. Another of our contributions is an executable model of this MAL that summarizes what the developers of Pip assume about it. It allows for making proofs but also for simulating inside Coq the execution of Pip.

Our last contribution is a formal, yet readable, formalization of the security features provided by Pip to applications, essentially memory isolation, and consistency properties that are discovered while making the proof of isolation.

Related work. There have been many efforts to make formal proofs about memory management in operating system kernels.

One of the most significant is the formal proof in the Isabelle/HOL proof assistant of the memory protection model of the microkernel seL4 [2]. The proof deals with an abstract model whose accuracy is then proved [6].

[1] `http://pip.univ-lille1.fr`

There is also the hypervisor CertiKOS [3] whose memory manager BabyVMM is formally verified by a series of refinements that are formalized in the Coq proof assistant [10]. In the same vein, a framework to formally verify preemptive kernels by refinement is proposed in [12].

In contrast to those works above, we are not making our proof on a model of a kernel, but directly on its source code that is written in Gallina thus making irrelevant the question of accuracy of the model.

In [1] and [5], abstract models of kernels, but not their implementations, were considered. On the contrary, in this deliverable, we do not deal with a model of the Pip protokernel but directly with its source code. Moreover, in [1] memory isolation was proved form the point of view information flow while here were are treating memory isolation at the lower level of page table management (information access).

In [8], the functional correctness of the memory manager of a microkernel was proved, but no memory isolation properties were proved. Moreover, this memory manager lives in a higher layer of the operating system than the lower-level layer we assume in this deliverable.

It is shown in [4] how an operating system can be entirely written in the functional programming language Haskell. In order to allow this system to manage physical memory, access hardware devices and execute processes is user mode, a monadic low-level interface was developed. While this approach demonstrates the feasibility of implementing an OS using a functional language, it introduces new limitations for formal verification such as including the garbage collector into the trusted computing base. In the same vein, MirageOS is a library operating system based on the OCaml language [7]. It runs under the Xen hypervisor whereas Pip runs on bare metal. Those works differ from Pip in two different ways. First, the monadic low-level interface — the Memory Abstraction Layer — on which Pip relies is much thinner and closer to the hardware than in those works. Second, although Pip is written in a functional language, it is written with a state monad allowing for an imperative style programming that can be directly translated into C, thus avoiding the use of a garbage collector.

1.2 Scope of this document

After some background on the Pip protokernel in Section 2, we present in Section 3 our executable model of the MAL. We show in Section 4 how Pip is coded directly in Gallina. We finally present in Section 5 the specification of memory isolation and the other invariants that must be preserved by Pip for ensuring isolation.

1.3 Related documents

2 The isolation manager

Pip is a protokernel: it allows for kernels, ranging from hypervisors to monolithic kernels, to be developped as user mode applications. This means that only the code of Pip is executed in kernel mode (i.e., the privileged mode of the hardware). Indeed, code running in kernel mode has direct access to the whole memory and hardware. It is thus clearly better, from a security point of view, to keep this code as minimal as possible. This

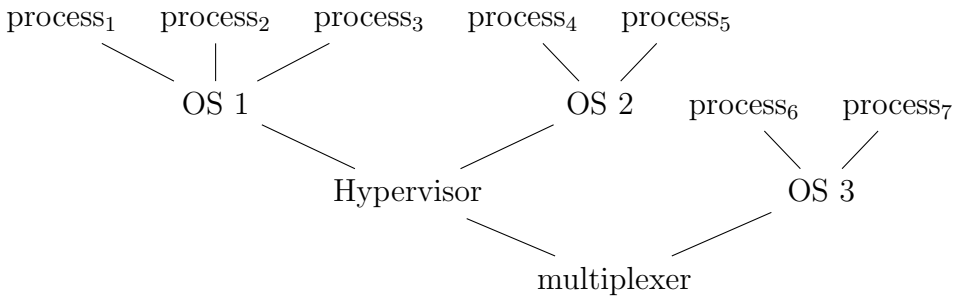

Figure 1: An example of partition tree

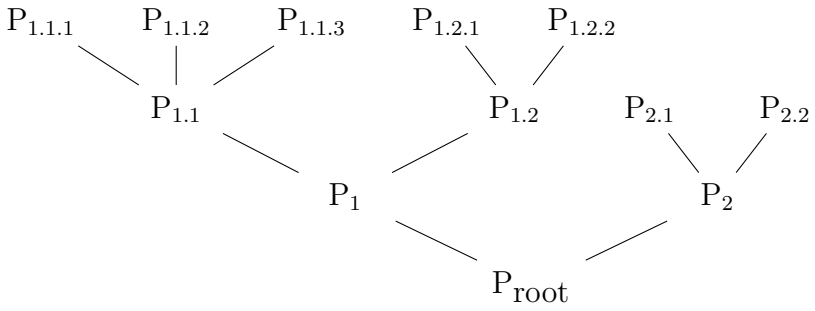

Figure 2: The view by Pip of the partition tree from Fig. 1

stems from the general principle that the trusted computing base (TCB) should be kept minimal. Pip only provides system calls for the management of partitions of the memory and for the basic dealing of control flow, thus reducing the trusted computing base to its bare minimum. More precisely, Pip provides 9 services that can be called from any partition. Only two of them are for basic control flow, the remaining ones are for memory management. For instance, Pip provides services to create or delete child partitions.

Partition tree. Fig. 1 illustrates how Pip can be used to partition the available memory: all the available memory makes up a *root* partition. That partition is named "multiplexer" in the figure. The code running in a partition can use the memory management services exposed by Pip to create a new *child* partition by lending some of its own pages to this new partition. That is how the "multiplexer" can create two child partitions, "Hypervisor" and "OS 3"; and how the partition "Hypervisor" can in turn share out its pages to other child partitions, "OS 1" and "OS 2". All the partitions created will thus form a tree.

Note that the "multiplexer" partition (and every partition with child partitions) is responsible of both sharing its memory between its child partitions but also scheduling them (it can implement any scheduling policy). Recall that the multiplexer is in a partition and is thus executed in user mode while traditionally (even in the case of an exokernel) the scheduler is part of the code that is executed in kernel mode. As we said before, this design choice comes from Pip's goal to keep the TCB minimal.

However, from the point of view of Pip which is the one adopted in this deliverable, partitions are just pieces of memory whose exact content is not relevant: only their relations in the partition tree is relevant for Pip. This is illustrated Fig. 2: Pip cares only

about the tree structure, not about what code is running in a given partition.

When a parent partition creates a child partition, it lends some of its memory to the child for its own use and but it also lends some to Pip, which it uses to store data related to this new partition. While the parent partition can read and write in the memory of its child partition (we call this property *vertical sharing*), it cannot access anymore to the pages lent to the kernel for security reasons (we want to prevent a partition from messing up Pip data structures). Sibling partitions cannot access each other's memory (we call this property *horizontal isolation*). For example, thanks to horizontal isolation, the partition "Hypervisor" in Fig. 1 is disjoint from the partition "OS 3". Thus, code running in the partition "Hypervisor" cannot access the memory in the partition "OS 3". Moreover, since partitions "OS 1" and "OS 2" are included in the partition "Hypervisor", they are also disjoint from the partition "OS 3". However their parent partition "Hypervisor" can access their memory and the memory of their descendant partitions $\text{process}_1, \ldots, \text{process}_5$ thanks to vertical sharing.

Data structures. A partition is identified by a physical page number so-called the partition descriptor. It contains required data to access to the whole data structure. The physical pages, lent to Pip to manage a partition, are organized in three tree-like structures: the MMU tables, the first shadow and the second shadow. As the name suggests, the first and the second shadows structures follow the same hierarchy as the MMU tables. The aim of this organization is to keep additional information about each physical page lent to a partition. Those pages are for the partition's own use and we call them mapped pages. The root of each tree structure is stored into the partition descriptor. Indeed, Pip explores the MMU tables tree structure to return the mapped page associated to a given virtual address. Relying on the same virtual address, Pip explores the first shadow tree structure to check if the mapped page represents a partition descriptor of a child or to check evenly if this mapped page is already attributed to a child. Likewise, to return back this page into the parent of this partition, Pip uses the same virtual address to explore the second shadow tree structure and get the virtual address of the mapped page into the parent virtual space. Note that, once the mapped pages are lent to children, the kernel keeps their mapping unchanged into the parent.

3 Model in Coq of the memory hardware and access operations

Pip's core is implemented in Gallina, which is a purely functional language so it was written using a *monad* to explicit the sequences of operations. In order to perform the actual basic operations, it is built over a thin layer of C and assembly code, called Memory Abstraction Layer (MAL), providing the set of functions necessary to access the memory (reading and writing values in physical memory), to configure the MMU and to change the current mode (set it to user mode). That layer was design to also hide the differences between various architectures.

In order to reason about Pip's core, we therefore built a model in Coq of the MAL that is not bound to a particular architecture. We assume in the proofs of the core that the actual implementations of the MAL abides by this model.

3.1 Hardware state

From the point of view of Pip, the relevant part of the hardware state consists of the partition descriptor of the partition that is currently active and of the parts of the physical memory where Pip stores its own data:

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

Any attempt by Pip to access a physical address that is not explicitly defined in the `memory` part of the state is considered in the model as an undefined behavior, and, of course, we will have to prove that Pip never exhibits any undefined behaviors. The physical memory is defined as an association list which maps relevant physical addresses to values. The keys for this association list will thus be physical addresses of type `paddr`. A physical address consists of a physical page number and a position into this page:

```
Definition paddr := page * index.
```

where `page` and `index` are positive integers with a certain bound and are defined as dependent records. A physical page number is a positive integer that is less than the number of pages `nbPage` which is a parameter of the hardware architecture:

```
Record page := { p :> nat ; Hp : p < nbPage }.
```

Likewise, an index is a positive integer that is less than the size `tableSize` of a table, which is also a parameter of the hardware architecture:

```
Record index := { i :> nat ;  Hi : i < tableSize }.
```

It is very important to keep track in the model of the types of values stored by Pip to make sure that Pip does not contain any type confusion vulnerability: for instance, if Pip writes a virtual address at some physical address in the memory, it should later be read again as a virtual address. Otherwise, it should, and will, be considered an undefined behavior in the model. The different types of values that could be stored into physical memory by Pip are listed in the following definition of the sum type `value`:

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

where a value could be a physical entry of type `Pentry`, a virtual entry of type `Ventry`, a physical page number of type `page`, a virtual address of type `vaddr` or an index into a table of type `index`.

The type `Pentry` consists of a physical page number equipped of several flags of type `bool`.

```
Record Pentry : Type:= {
    read:    bool ;
    write:   bool ;
    exec:    bool ;
    present: bool ;
    user:    bool ;
    pa:      page }.
```

Values of the above type are used by Pip to configure the MMU tables of partitions in order to manage their virtual memory. For instance, the the flag `user` indicates whether the associated physical page `pa` is accessible in the user mode.

Similarly, the type `Ventry` consists of a virtual address (of type `vaddr`) equipped with a boolean flag.

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

As detailed in Section 2, a shadow data structure follows the same hierarchy as the MMU tables in order to provide additional information about each mapped page. Thus, the values of type `Ventry` are used to configure the tables of the first shadow structure that correspond to the lowest MMU level. The flag `pd` indicates if the associated physical page number corresponds to a child of this partition. The virtual address `va` is used to check if this physical page is already attributed to a child.

A virtual address `vaddr` is modeled as a list of indices of length `nbLevel` plus one where `nbLevel` is the number of levels of the MMU in the considered hardware architecture.

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

The last index of the list corresponds to the offset of the corresponding physical address.

3.2 Model of the MAL

Concretely, the MAL is a C and assembly library that implements the architecture-dependent part of Pip and should be ported to support new platforms. In all cases, it implements a minimal interface that is limited to accesses to physical memory in order to configure the data structures of partitions, virtual memory activation and a set of operations for data processing like comparing two virtual addresses stored in memory. It consists, especially, of bitwise operations of constant computational complexity.

In this section, we focus on its model in Coq that is not bound to any particular architecture.

State Monad. Gallina, the specification language of Coq, is a purely functional language. But in order to implement the kernel services of Pip, we need to provide imperative features to access the current state and possibly update it. For that purpose, we will use a state monad that can be seen as an abstract data type that wraps updating state and undefined behaviors [9, 11]. It gives us a way to use side effects in the pure functional language of Coq. We define the state monad abstract type `LLI A` as follows:

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

where `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 behavior.

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

In order to arrive at a monad, we also define the standard operations `ret` and `bind`. And to make it a state monad we also define additional operations `get` and `put` that respectively returns the current state and update it. Of course, those last two operations will not be part of the API because they would not have any meaningful counterparts in the C implementation of the MAL. They are used here to model the behavior of the primitives exported by the API as we will see in the following examples of primitives.

We denote `perform x := m in e` the monadic operation `bind m (fun x ⇒ e)` that performs two actions `m` and `e` sequentially.

We first consider examples of primitives in the API that concern physical memory accesses. More precisely, we will look at examples of primitives for reading and writing a virtual address in physical memory.

We begin with the primitive `writeVirtual`:

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

The first and second argument of `writeVirtual` correspond respectively to a physical-page number and a position in this page. The third argument represents the value of the virtual address to store at the given physical address. Indeed, `writeVirtual` modifies the association list representing the physical memory in the state by using the `modify` operation of the state monad to associate the virtual address value to that physical address in the association list. The operation `modify` is of type `(state → state) → LLI unit`. `modify f` retrieves the current state `s` by using the `get` operation of type `LLI state` and updates it with the new state `f s` using the `put` operation of type `state → LLI unit`.

The `readVirtual` primitive takes two arguments that represent the memory location in which a virtual address is supposedly already saved. To read the virtual address from physical memory of the current state, we start by checking the type of the value into this location.

```
Definition readVirtual (p : page) (idx : index) : LLI vaddr :=
    perform s := get in
    let entry := lookup p idx s.(memory) beqPage beqIndex in
    match entry with
    | Some (VA v) ⇒ ret v
    | Some _      ⇒ undefined 3
    | None        ⇒ undefined 2
    end.
```

The above primitive make sure that, first, there is a value at this physical address, and, second, that this value is a virtual address. If those conditions are not met, then this is considered an undefined behavior in the model.

Finally, the API exports functions to process data. There is, for example, the operation `succ` that computes the successor of a given index. This is in the API of the MAL because it depends on the architecture:

```
Program Definition succ (n : index) : LLI index :=
    let isucc := n+1 in
    if lt_dec isucc tableSize
```

```
  then ret (Build_index isucc _)
  else undefined 28.
```

If the new index value exceeds the table size (fixed by the hardware architecture), then this is an undefined behavior in the model. Similarly, for each kernel data type, we define all required operations for efficient kernel data processing.

It is interesting to note that different functions in the API can be implemented by the same C function. For example, functions for writing in physical memory a virtual address `writeVirtual`, a virtual entry `writeVirEntry` or an index `writeIndex` are implemented by the same C function `writePhysical`. The only difference between those function is their type which is irrelevant in C while it is highly relevant in the model in Coq.

4 Code in Coq of the protokernel Pip

Gallina without garbage collecting. Having a garbage collector (GC) is a standard feature of functional programming languages such as Gallina. However using such an automatic memory management for the development of Pip would require to include the GC into the TCB. But this would be antithetical to the purpose of Pip which is to provide proved memory isolation based on the smallest possible TCB.

Moreover, is spite of significant progress made in the development of GCs, modern ones still incur pauses throughout the execution for collecting garbage. This is not acceptable in the context of kernel development because a kernel must deal swiftly with hardware interrupts when they occur. For instance, if the network card raises a hardware interrupt to signal the arrival of a packet, it must be dealt with straight away and the kernel should thus not be paused because of garbage collection.

Still we want to use the functional language Gallina because it is native to Coq and thus its use brings all the advantages of a shallow embedding. In other words, we want to have the cake and eat it too.

We avoid garbage collecting by using an imperative style of coding that is so close to the code one would write in C that it can be literally translated into C. This is made possible by the usage of the state monad introduced in Section 3.2. We also exclude the use of all objects that would require a GC such as the inductive type `list` of the standard library of Coq or any other recursive inductive type. When we need a list — and indeed Pip uses lists, but also trees — we explicitly encode it in physical memory, exactly as it would have to be done using `struct` in C; this means in particular that Pip manipulates explicitly the addresses (pointers) of the cells of a list when visiting or modifying the list. This requires reading and updating the memory at specific addresses which is the main purpose of the API of the MAL (cf. Section 3.2).

Here, we present, as a simple example, the internal function `getFstShadow` and its translation into C. It consists in calling sequentially three primitives of the MAL that are `getSh1idx`, `succ` and `readPhysical`. It returns the first shadow page of a given partition.

```
Definition getFstShadow partition:=                                    1
  perform idx := getSh1idx in                                          2
  perform idxSucc := MALInternal.Index.succ idx in                     3
  readPhysical partition idxSucc.                                      4
```

The binding of variables in Gallina is naturally translated into constant assignment in C.

```
uintptr_t getFstShadow(const uintptr_t partition) {                    1
  const uint32_t idx = getSh1idx();                                    2
  const uint32_t idxSucc = succ(idx);                                  3
  return readPhysical(partition, idxSucc); }                           4
```

Recursive functions. Among the difficulties encountered in the coding of Pip in Coq, there are the limitations imposed by Coq on recursive functions because functions defined in Coq must always terminate. This is enforced by syntactical restrictions to ensure that a certain argument is always *smaller* in recursive calls: the simplest way is for this argument to be of an inductive type and for the recursive call to be on a subterm (so-called structural recursion). Unfortunately, we cannot directly apply this simple method for Pip code for mainly two reasons.

First, in the case the recursive argument is of an inductive type in the model of the MAL (for example, when this argument is an index, that is modeled as a value of inductive type `nat` with a side condition), it is necessarily an abstract data type because Coq's inductive types have no counterpart in C. Moreover, in most cases, this index argument will not decrease but increase at each recursive call. This is for instance the case when Pip goes through a table starting at the index 0 (see, for instance, the function `initVEntryTableAux` in Appendix A).

Second, data structures used by Pip such as lists and trees are encoded in memory with pointers. Therefore, going through these structures recursively cannot be seen as terminating by Coq. Indeed, there is no guarantee when we defined those functions that their arguments will be well-formed lists or trees without loop (see, for instance, the function `writeAccessibleRec` in Appendix B).

Our solution is to rely on the fact that the number of recursive calls for any function in Pip is always bounded. These bounds can be computed based on the parameters of the hardware architecture such as the size of a memory page. Obviously, this means that we need to prove that the computed bound is large enough.

Simulation of Pip inside Coq. On the one hand, the C code will be linked to the C library implementing the API of the MAL for the targeted architecture. On the other hand, the code can be executed inside Coq for simulation purposes. This is indeed possible because our model of the MAL described in Section 3 is executable. Indeed, before trying to prove anything on Pip's code, we started by debugging it to fix some initial errors and weaknesses. The simulation allows to determine the state of physical memory during the execution of services. For simulation, we first need to choose values for the hardware architecture parameters. To that end, we considered an MMU with two levels and kernel configuration tables with 12 entries (minimal table size to configure a single partition). The initial state corresponded to the initialization of the first partition virtual space with some mapped pages. As to be expected with such code manipulating complex data structures in memory, we found and fixed bugs in the initial code.

5 Isolation and consistency formalization

A formal proof is not enough. One has to be sure that what is formally specified is really what one wants to prove. To this end, we make sure that our formal specification of memory isolation is understandable by using readable definitions. In this section, we start with a short description about Hoare logic then we focus on the formalization of horizontal and kernel isolation and vertical sharing properties and we introduce the consistency properties required for isolation properties to be preserved.

5.1 Hoare logic on top of state monad

Our verification approach builds on reasoning about invariants. Thus, we define a Hoare logic on top of our state monad that allows specifying properties about the services of the kernel. In coq, we define the Hoare triple 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.
```

where: `P` is a precondition, i.e. a unary predicate on the starting state; `m` is a computation returning a result of type `A`, i.e. the computation `m` is of type `LLI A`; `Q` is a postcondition, i.e. a binary predicate on the returned value and on the ending state.

By definition, `hoareTriple P m Q` holds iff: for all state `s`, if `P` holds for `s` then performing `m` in state `s` will yield a value `a` and end in a state `s'` such that the postcondition `Q` holds for `a` and `s'`. If performing `m` yield an undefined behavior, the triple does not hold. As a result, not only security properties are involved into our formal proof but some functional-correctness properties of Pip are proved: the services of Pip never lead to an undefined behavior.

5.2 Isolation and vertical sharing properties

The first property is the isolation between children. Indeed, if `child1` and `child2` are two different children of a given partition `parent`, then all the pages which are used by `child1` are different from all pages used by `child2`. The `partitionsIsolation` is defined as following:

```
Definition partitionsIsolation (s: state) : Prop :=
  ∀ parent child1 child2: page, parent ∈ getPartitions s →
  child1 ∈ getChildren parent s → child2 ∈ getChildren parent s →
  child1 ≠ child2 → disjoint (getUsedPages child1 s) (getUsedPages child2 s).
```

where:

- `getPartitions` is a function returning the list of all existing partitions of a given state; this list is very naturally built by a search of the tree of partitions.
- `getUsedPages` returns the list of all the pages *used* by a partition; by "used", we mean all the mapped pages of the partition but also all the pages lent to Pip to

manage the partition. That list is built by exploring the data structures associated to the partition.

Note that the above functions are only used for the specification of Pip but not in its code. We will introduce similar functions in the rest of this deliverable.

The next property defines the vertical sharing relation between a parent partition and its children. The property `verticalSharing` holds of a state `s` iff for all `child` is in the list of children of a given partition `parent`, then all pages used by `child` are mapped into `parent`.

```
Definition verticalSharing (s: state) : Prop :=
  ∀ parent child: page, parent ∈ getPartitions s →
                        child  ∈ getChildren parent s →
                        getUsedPages child s ⊆ getMappedPages parent s.
```

where `getMappedPages` returns the list of mapped pages of a given partition.

The last property is about pages lent to the kernel to manage partitions. Indeed, these pages should not be accessible by any partition. The `kernelDataIsolation` property ensures that for any state `s` of the system, if `partition1` and `partition2` are two partitions, then the code running in `partition2` cannot write in the pages containing `partition1` configuration tables (*i.e.* the pages lent to Pip to manage `partition1`).

```
Definition kernelDataIsolation (s: state) : Prop := ∀ partition1 partition2,
  partition1 ∈ getPartitions s → partition2 ∈ getPartitions s →
  disjoint (getAccessibleMappedPages partition1 s) (getConfigPages partition2 s).
```

where `getAccessibleMappedPages` returns all mapped pages marked as accessible in a given partition.

5.3 Consistency

The consistency properties are mandatory to prove the properties previously detailed. These properties are discovered along the proof of isolation. Since the proof is not yet finished, the list of properties is probably incomplete. We split the consistency properties into three categories. In this section, we illustrate each category with one property.

- Well-typedness: The first category is about the kernel data types. As detailed in Section 3, to avoid type confusion, we have to keep track in the model of the types of values stored by Pip into its different data structures. To ensure that, we define the property `dataStructurePdSh1Sh2asRoot`. This property uses data types introduced in Section 2.

  ```
  Definition dataStructurePdSh1Sh2asRoot (idxroot: index) (s: state) :=
    ∀ partition: page, partition ∈ getPartitions s →
    ∀ rootStruct: page, nextEntryIsPP partition idxroot rootStruct s →
    ∀ stop: nat, ∀ table: page, ∀ idx: index,
    ∀ va: vaddr, getIndirection rootStruct va nbLevel stop s = Some table →
    ( table = defaultPage ∨
      ( ( ( stop < nbLevel ∧ isPE table idx s ) ∨
          ( stop = nbLevel ∧ ( (isVE table idx s ∧ idxroot = sh1idx) ∨
                               (isVA table idx s ∧ idxroot = sh2idx) ∨
  ```

```
                                (isPE table idx s ∧ idxroot = PDidx) ))) ∧
    table ≠ defaultPage ) ).
```

The parameter `idxroot` indicates which data structure is involved. The `PDidx`, `sh1idx` and `sh2idx` correspond, respectively, to the position of the roots of the MMU tables, the first shadow and the second shadow data structures into the partition descriptor. All these positions are defined by the MAL.

To explore a tree structure path, from the root to a leaf, we defined the function `getIndirection`. It takes as parameters the root of the tree structure `rootStruct`, the virtual address `va` to explore, the number of levels of the MMU (*i.e.* the maximal height of the tree structure) and a bound `stop` that corresponds to the depth of the table to explore, and returns the table at depth `stop`. This table could be a leaf if the value of `stop` is equal to `nbLevel`, and a node otherwise.

In the case of the second shadow, its data structure is well typed if, for any partition `partition` into the partition tree, the table returned by the function `getIndirection` contains either: values of type `vaddr` if it is associated to the lowest MMU level, denoted as `stop = nbLevel`; or values of type `Pentry` if it corresponds to a node.

- Partition tree structure: This category includes all the properties about the partition tree structure and we choose to detail the `noCycleInPartitionTree` property. This property is required to ensure that there is no cycle in the partition tree of a given state of the system. Indeed, this property holds of a state `s`, if for all partition `partition` in the partition tree, and for all partition `ancestor` which is an ancestor of `partition` then `ancestor` and `partition` are different.

```
Definition noCycleInPartitionTree s :=
  ∀ ancestor partition, partition ∈ getPartitions s →
                        ancestor ∈ getAncestors partition s →
                        ancestor ≠ partition.
```

- Flag semantics: This category focuses on the signification of each flag used in the data structures. For instance, the property `isPresentNotDefaultIff` precises that if the flag `present`, used in the type `Pentry`, is equal to `false` then the associated physical page number is equal to the default value `defaultPage`, defined by the MAL.

```
Definition isPresentNotDefaultIff s :=
  ∀ table idx, readPresent table idx (memory s) = Some false ⟷
               readPhyEntry table idx (memory s) = Some defaultPage.
```

- Properties about pages: This category concerns multiple properties about mapped and kernel pages and relations between them. For example, the `noDupMappedPagesList` requires that the mapped pages of a single partition are all different.

```
Definition noDupMappedPagesList s (partition: page) :=
  partition ∈ getPartitions s → NoDup (getMappedPages partition s).
```

References

[1] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Jesús Mauricio Chimento, and Carlos Luna. Formally verified implementation of an idealized model of virtualization. In Ralph Matthes and Aleksy Schubert, editors, *19th International Conference on Types for Proofs and Programs, TYPES 2013, April 22-26, 2013, Toulouse, France*, volume 26 of *LIPIcs*, pages 45–63. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

[2] Dhammika Elkaduwe, Gerwin Klein, and Kevin Elphinstone. Verified protection model of the sel4 microkernel. In Natarajan Shankar and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments, Second International Conference, VSTTE 2008, Toronto, Canada, October 6-9, 2008. Proceedings*, volume 5295 of *Lecture Notes in Computer Science*, pages 99–114. Springer, 2008.

[3] Liang Gu, Alexander Vaynberg, Bryan Ford, Zhong Shao, and David Costanzo. Certikos: a certified kernel for secure cloud computing. In Haibo Chen, Zheng Zhang, Sue Moon, and Yuanyuan Zhou, editors, *APSys '11 Asia Pacific Workshop on Systems, Shanghai, China, July 11-12, 2011*, page 3. ACM, 2011.

[4] Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew P. Tolmach. A principled approach to operating system construction in haskell. In Olivier Danvy and Benjamin C. Pierce, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, pages 116–128. ACM, 2005.

[5] Narjes Jomaa, David Nowak, Gilles Grimaud, and Samuel Hym. Formal proof of dynamic memory isolation based on MMU. In *10th International Symposium on Theoretical Aspects of Software Engineering, TASE 2016, Shanghai, China, July 17-19, 2016*, pages 73–80. IEEE Computer Society, 2016.

[6] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby C. Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.*, 32(1):2:1–2:70, 2014.

[7] Anil Madhavapeddy and David J. Scott. Unikernels: the rise of the virtual library operating system. *Commun. ACM*, 57(1):61–69, 2014.

[8] Nicolas Marti, Reynald Affeldt, and Akinori Yonezawa. Formal verification of the heap manager of an operating system using separation logic. In Zhiming Liu and Jifeng He, editors, *Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006, Macao, China, November 1-3, 2006, Proceedings*, volume 4260 of *Lecture Notes in Computer Science*, pages 400–419. Springer, 2006.

[9] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[10] Alexander Vaynberg and Zhong Shao. Compositional verification of a baby virtual memory manager. In Chris Hawblitzel and Dale Miller, editors, *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings*, volume 7679 of *Lecture Notes in Computer Science*, pages 143–159. Springer, 2012.

[11] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1992.

[12] Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. A practical verification framework for preemptive OS kernels. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 59–79. Springer, 2016.

A An example of recursive function on an abstract data type

The `initVEntryTable` function initializes virtual entries `VEntry` of a given table `shadow1` by default value (`defaultVAddr` for virtual address `va` and `false` for the `pd` flag.

```
Fixpoint initVEntryTableAux timeout shadow1 idx :=
  match timeout with
  | 0 ⇒ ret tt
  | S timeout1 ⇒
    perform maxindex := getMaxIndex in
    perform res := MALInternal.Index.ltb idx maxindex in
    if (res)
    then
      perform defaultVAddr := getDefaultVAddr in
      writeVirEntry shadow1 idx defaultVAddr;;
      perform nextIdx :=  MALInternal.Index.succ idx in
      initVEntryTableAux timeout1 shadow1 nextIdx
    else
      perform defaultVAddr := getDefaultVAddr in
      writeVirEntry shadow1 idx defaultVAddr
  end.
```

The `initVEntryTable` function fixes the timeout value of `initVEntryTableAux`:

```
Definition initVEntryTable shadow1 idx :=
  initVEntryTableAux tableSize  shadow1 idx.
```

The above recursive function is translated in C in the following way:

```
void initVEntryTableAux(const unsigned timeout, const uintptr_t shadow1, const uint32_t idx)
{
        if ((timeout == 0)) {
                ;
        } else {
```

```
                const unsigned timeout1 = (timeout - 1);
                const uint32_t maxindex = getMaxIndex();
                const int res = ltb(idx, maxindex);
                if (res) {
                        const uintptr_t defaultVAddr = getDefaultVAddr();
                        writeVirEntry(shadow1, idx, defaultVAddr);
                        const uint32_t nextIdx = succ(idx);
                        initVEntryTableAux(timeout1, shadow1, nextIdx);
                } else {
                        const uintptr_t defaultVAddr = getDefaultVAddr();
                        writeVirEntry(shadow1, idx, defaultVAddr);
                }
        }
}

void initVEntryTable(const uintptr_t shadow1, const uint32_t idx)
{
        initVEntryTableAux(tableSize, shadow1, idx);
}
```

B Going through a tree encoded in memory with pointers

The `writeAccessibleRec` function updates the user access flag of a mapped physical page which corresponds to a given virtual address `va` in all ancestors of a given partition `descParent`.

```
Fixpoint writeAccessibleRec timout (va : vaddr) (descParent : page) (flag : bool):=
match timout with
| 0 ⇒ ret false
| S timout1 ⇒
perform root := getMultiplexer in
perform isMultiplexer := MALInternal.Page.eqb descParent root in
if isMultiplexer (** stop if parent is the root *)
then ret true
else
(** get the snd shadow of the parent to get back the virtual address into the first ancestor *)
perform sh2Parent := getSndShadow descParent in
perform L:= getNbLevel in
perform idx := getIndexOfAddr va fstLevel in
perform ptsh2 := getTableAddr sh2Parent va  L in
perform isNull := comparePageToNull ptsh2 in
if isNull then ret false else
(** read the virtual address into the first ancestor *)
perform vaInAncestor := readVirtual ptsh2 idx in
(** get the first ancestor partition descriptor *)
perform ancestor := getParent descParent in
(** get the page directory of the ancestor partition descriptor *)
```

```
perform pdAncestor := getPd ancestor in
(** set access rights of the virtual address *)
  perform nullAddrV :=  getDefaultVAddr in
  perform res := MALInternal.VAddr.eqbList nullAddrV vaInAncestor in
  if (res )
  then ret false
  else
    perform pt := getTableAddr pdAncestor vaInAncestor L in
    perform isNull := comparePageToNull pt in
    if isNull then ret false else
    perform idx := getIndexOfAddr vaInAncestor fstLevel in
    writeAccessible pt idx flag ;;
(** recursion *)
writeAccessibleRec timout1 vaInAncestor ancestor flag
end.
```