Formally verifying isolation and availability in an idealized model of virtualization*

Gilles Barthe[1], Gustavo Betarte[2], Juan Diego Campo[2], and Carlos Luna[2]

[1] IMDEA Software, Madrid, Spain.
[2] InCo, Facultad de Ingeniería, Universidad de la República, Uruguay

Abstract. Hypervisors allow multiple guest operating systems to run on shared hardware, and offer a compelling means of improving the security and the flexibility of software systems. We formalize in the Coq proof assistant an idealized model of a hypervisor, and formally establish that the hypervisor ensures strong isolation properties between the different operating systems, and guarantees that requests from guest operating systems are eventually attended.

1 Introduction

Hypervisors allow several operating systems to coexist on commodity hardware, and provide support for multiple applications to run seamlessly on the guest operating systems they manage. Moreover, hypervisors provide a means to guarantee that applications with different security policies can execute securely in parallel, by ensuring isolation between their guest operating systems. In effect, hypervisors are increasingly used as a means to improve system flexibility and security, and authors such as [10] predict that their use will become ubiquitous in enterprise data centers and cloud computing.

The increasingly important role of hypervisors in software systems makes them a prime target for formal verification. Indeed, several projects have set out to formally verify the correctness of hypervisor implementations. One of the most prominent initiatives is the Microsoft Hyper-V verification project [8, 16], which has made a number of impressive achievements towards the functional verification of the legacy implementation of the Hyper-V hypervisor, a large software component that combines C and assembly code (about 100 kLOC of C and 5kLOC of assembly). The overarching objective of the formal verification is to establish that a guest operating system cannot observe any difference between executing through the hypervisor or directly on the hardware. The other prominent initiative is the L4.verified project [14], which recently completed the formal verification of the seL4 microkernel, a general purpose operating system

* Partially funded by European Project FP7 256980 NESSoS, Spanish project TIN2009-14599 DESAFIOS 10, Madrid Regional project S2009TIC-1465 PROMETIDOS and Project ANII-Clemente Estable PR-FCE-2009-1-2568 VirtualCert.

of the L4 family. The main thrust of the formal verification is to show that an implementation of the microkernel correctly refines an abstract specification.

Reasoning about implementations provides the ultimate guarantee that deployed hypervisors provide the expected properties. There are however significant hurdles with this approach, especially if one focuses on proving security properties rather than functional correctness. First, the complexity of formally proving non-trivial properties of implementations might be overwhelming in terms of the effort it requires; worse, the technology for verifying some classes of security properties may be underdeveloped: specifically, liveness properties are notoriously hard to prove, and there is currently no established method for verifying security properties involving two system executions, a.k.a. 2-properties [7], for implementations. Second, many implementation details are orthogonal to the security properties to be established, and may complicate reasoning without improving the understanding of the essential features for guaranteeing isolation among guest operating systems. Thus, there is a need for complementary approaches where verification is performed on idealized models that abstract away from the specifics of any particular hypervisor, and yet provide a realistic setting in which to explore the security issues that pertain to the realm of hypervisors.

This article initiates such an approach by developing a minimalistic model of a hypervisor, and by formally proving that the hypervisor correctly enforces isolation between guest operating systems, and under mild hypotheses guarantees basic availability properties to guest operating systems. In order to achieve some reasonable level of tractability, our model is significantly simpler than the setting considered in the Microsoft Hyper-V verification project, it abstracts away many specifics of memory management such as translation lookaside buffers (TLBs) and shadow page tables (SPTs) and of the underlying hardware and runtime environment such as I/O devices. Instead, our model focuses on the aspects that are most relevant for isolation properties, namely read and write resources on machine addresses, and is sufficiently complete to allow us to reason about isolation properties. Specifically, we show that an operating system can only read and modify memory it owns, and a non-influence property [18] stating that the behavior of an operating system is not influenced by other operating systems. In addition, our model allows reasoning about availability; we prove, under reasonable conditions, that all requests of a guest operating system to the hypervisor are eventually attended, so that no guest operating system waits indefinitely for a pending request. Overall, our verification effort shows that the model is adequate to reason about safety properties (read and write isolation), 2-safety properties (OS isolation), and liveness properties (availability).

Contents of the paper Section 2 provides a primer on virtualization, focusing on the elements that are most relevant for our formal model, which we develop in Section 3. Isolation properties are considered in Section 4, whereas availability is discussed in Section 5. Section 6 considers related work; further work and conclusions are presented in Section 7. The formal development is available at `http://www.fing.edu.uy/inco/grupos/gsi/proyectos/virtualcert.php`, and can be verified using the Coq proof assistant (version 8.2) [21].

Notation We use standard notation for equality and logical connectives. We extensively use record types, enumerated types, and (parametric) sum types. Record types are of the form $\{l_1 : T_1, \ldots, l_n : T_n\}$, whereas their elements are of the form $\langle t_1, \ldots, t_n \rangle$. Field selection is abbreviated using dot notation. Enumerated types and parametric sum types are defined using Haskell-like notation; for example, we define for every type T the type $\mathit{option}\ T \stackrel{\text{def}}{=} \mathit{None} \mid \mathit{Some}\ (t : T)$. We also make an extensive use of partial maps: the type of partial maps from objects of type A into objects of type B is written $A \mapsto B$. Application of a map m on an object of type a is denoted $m[a]$ and map update is written $m[a := b]$, where b overwrites the value, if any, associated to a. Finally, runs are modeled co-inductively, using streams. The type of streams of type A is written $[A]_\infty$. Objects of type $[A]_\infty$ are constructed with the (infix) operator ::, hence $x :: xs$ is of type $[A]_\infty$ whenever x is of type A and xs is of type $[A]_\infty$. Given $s : [A]_\infty$ we let $s[i]$ denote the i-th element of s.

2 A primer on virtualization

Virtualization is a technique used to run on the same physical machine multiple operating systems, called *guest operating systems*. The hypervisor, or Virtual Machine Monitor [11], is a thin layer of software that manages the shared resources (e.g. CPU, system memory, I/O devices). It allows guest operating systems to access these resources by providing them an abstraction of the physical machine on which they run. One of the most important features of a virtualization platform is that its OSs run isolated from one another. In order to guarantee isolation and to keep control of the platform, a hypervisor makes use of the different execution modes of a modern CPU: the hypervisor itself and trusted guest OSs run in supervisor mode, in which all CPU instructions are available; while untrusted guest operating systems will run in user mode in which privileged instructions cannot be executed.

Historically there have been two different styles of virtualization: *full virtualization* and *paravirtualization*. In the first one, each virtual machine is an exact duplicate of the underlying hardware, making it possible to run unmodified operating systems on top of it. When an attempt to execute a privileged instruction by the OS is detected the hardware raises a trap that is captured by the hypervisor and then it emulates the instruction behavior. In the paravirtualization approach, each virtual machine is a simplified version of the physical architecture. The guest (untrusted) operating systems must then be modified to run in user CPU mode, changing privileged instructions to hypercalls, i.e. calls to the hypervisor. A hypercall interface allows OSs to perform a synchronous software trap into the hypervisor to perform a privileged operation, analogous to the use of system calls in conventional operating systems. An example use of a hypercall is to request a set of page table updates, in which the hypervisor validates and applies a list of updates, returning control to the calling OS when this is completed.

In this work, we focus on the memory management policy of a paravirtualization style hypervisor, based on the Xen virtualization platform [6]. Several features of the platform are not yet modeled (e.g. I/O devices, interruption system, or the possibility to execute on multi-cores), and are left as future work.

3 The model

In this section we present and discuss the formal specification of the idealized model. We first introduce the set of states, and the set of actions; the latter include both operations of the hypervisor and of the guest operating systems. The semantics of each action is specified by a precondition and a postcondition. Then, we introduce a notion of valid state and show that state validity is preserved by execution. Finally, we define execution traces.

3.1 Informal overview of the memory model

The most important component of the state is the memory model, which we proceed to describe. As illustrated in Figure 1, the memory model involves three types of addressing modes and two address mappings: the machine address is the real machine memory; the physical memory is used by the guest OS, and the virtual memory is used by the applications running on an operating system.

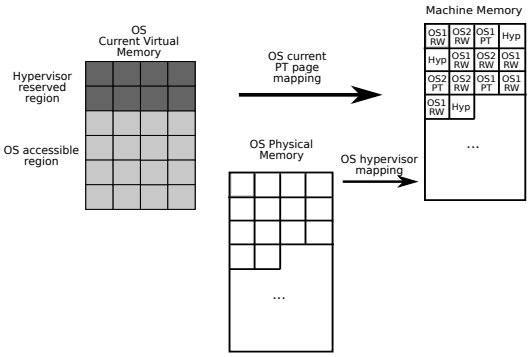

Fig. 1. Memory model of the platform

The *virtual memory* is the one used by applications running on OSs. Each OS stores a partial mapping of virtual addresses to machine addresses. This will allow us to represent the translation of the virtual addresses of the applications executing in the OS into real hardware addresses. Moreover, each OS has a designated portion of its virtual address space (usually abbreviated VAS) that is reserved for the hypervisor to attend hypercalls. We say that a virtual address *va* is *accessible* by the OS if it belongs to the virtual address space of the OS which is not reserved for the hypervisor. We denote the type of virtual addresses by *vadd*.

The *physical memory* is the one addressed by the kernel of the guest OS. In the Xen [6] platform, this is the type of addresses that the hypervisor exposes to the domains (the untrusted guest OSs in our model). The type of physical addresses is written *padd*.

The *machine memory* is the real machine memory. A mechanism of page classification was introduced in order to cover concepts from certain virtualization platforms, in particular Xen. The model considers that each machine address

that appears in a memory mapping corresponds to a memory page. Each page has at most one unique owner, a particular OS or the hypervisor, and is classified either as a data page with read/write access or as a page table, where the mappings between virtual and machine addresses reside. It is required to register (and classify) a page before being able to use or map it. The type of machine addresses is written *madd*.

As to the mappings, each OS has an associated collection of page tables (one for each application executing on the OS) that map virtual addresses into machine addresses. When executed, the applications use virtual addresses, therefore on context switch the current page table of the OS must change so that the currently executing application may be able to refer to its own address space. Neither applications nor untrusted OSs have permission to read or write page tables, because these actions can only be performed in supervisor mode. Every memory address accessed by an OS needs to be associated to a virtual address. The model must guarantee the correctness of those mappings, namely, that every machine address mapped in a page table of an OS is owned by it.

The mapping that associates, for each OS, machine addresses to physical ones is, in our model, maintained by the hypervisor. This mapping might be treated differently by each specific virtualization platform. There are platforms in which this mapping is public and the OS is allowed to manage machine addresses. The physical-to-machine address mapping is modified by the actions `page_pin` and `page_unpin`, as shall be described in Section 3.3.

3.2 Formalizing states

The platform state consists of a collection of components that we now proceed to describe.

Operating systems We start from a type *os_ident* of identifiers for guest operating systems, and a predicate *trusted_os* indicating whether a guest operating system is trusted. The state contains information about each guest OS current page table, which is a physical address, and information on whether it has a hypercall pending to be resolved. Formally the information is captured by a mapping *oss_map* that associates OS identifiers with objects of type *os*, where $os \stackrel{\text{def}}{=} \{curr\_page : padd, hcall : option\ Hyper\_call\}$, and $oss\_map \stackrel{\text{def}}{=} os\_ident \mapsto os$.

Execution modes Most hardware architectures distinguish at least two execution modes, namely *user mode* (usr) and *supervisor mode* (svc). These modes are used as a protection mechanism, where *privileged* instructions are only allowed to be executed in supervisor mode. In our model, untrusted OSs execute in user mode while trusted ones and the hypervisor execute in supervisor mode. When an untrusted OS needs to execute a privileged operation, it requests the hypervisor to do it on its behalf. Execution modes are formalized by the enumerated type *exec_mode*, where $exec\_mode \stackrel{\text{def}}{=} usr \mid svc$.

Moreover, there is a single active OS in each state. After requesting the hypervisor to execute some service, the active guest OS will turn in processor execution mode *waiting* until the service is completed and the execution control returned, switching then its execution mode to *running*. Active OS execution mode is formalized by the type $os\_activity \stackrel{\text{def}}{=} running \mid waiting$.

Memory mappings The mapping that, given an OS returns the corresponding mapping from physical to machine addresses, is formalized as an object of the type *hypervisor_map*, where $hypervisor\_map \stackrel{\text{def}}{=} os\_ident \mapsto (padd \mapsto madd)$. The real platform memory is formalized as a mapping that associates to a machine address a page, thus $system\_memory \stackrel{\text{def}}{=} madd \mapsto page$. A page consists of a page content and a reference to the page owner. Page contents can be either (readable/writable) values, an OS page table or nothing; note that a page might have been created without having been initialized, hence the use of option types. Page owners can be the hypervisor, a guest OS or none. Formally:

$content \stackrel{\text{def}}{=} RW\ (v : option\ Value) \mid PT\ (va\_to\_ma : vadd \mapsto madd) \mid Other$

$page\_owner \stackrel{\text{def}}{=} Hyp \mid Os\ (osi : os\_ident) \mid No\_Owner$

$page \stackrel{\text{def}}{=} \{page\_content : content, page\_owned\_by : page\_owner\}$

States The states of the platform are modeled by a record with six components:

$$\begin{aligned} State \stackrel{\text{def}}{=} \{\ & active\_os : os\_ident, \\ & aos\_exec\_mode : exec\_mode, \\ & aos\_activity : os\_activity, \\ & oss : oss\_map, \\ & hypervisor : hypervisor\_map, \\ & memory : system\_memory\} \end{aligned}$$

The component *active_os* indicates which is the active operating system and *aos_exec_mode* and *aos_activity* the corresponding execution and processor mode. The component *oss* stores the information of the guest operating systems of the platform. *hypervisor* and *memory* are the mappings used to formalize the memory model described in the previous section.

In the sequel, we use the following notation. Given states s and s', we define $s \sim_{map,idx} s'$ to be the relation that establishes that s and s' differ at most in the value associated to the index *idx* of the component *map* in the state s', and $s' = s.[c_1, \ldots, c_n]v_1, \ldots, v_n$ the relation that establishes that s and s' differ at most in the values $v_1, \ldots, v_n$ of the components $c_1, \ldots, c_n$ in state s'. Moreover, we define the predicate $os\_accessible(va)$, that holds if va belongs to the set of virtual addresses accessible by any OS.

Valid state We define a notion of valid state that captures essential properties of the platform. Formally, the predicate *valid_state* holds on state s if s satisfies

$$
\begin{array}{l}
\mathit{Pre}\ s\ (\texttt{read}\ va) \stackrel{\text{def}}{=} \mathit{os\_accessible}(va)\ \wedge\ s.\mathit{aos\_activity}\ =\ \mathit{running}\ \wedge \\
\quad \exists\ ma : \mathit{madd},\ \mathit{va\_mapped\_to\_ma}(s, va, ma)\ \wedge \\
\quad \mathit{is\_RW}((s.\mathit{memory}[ma]).\mathit{page\_content}) \\
\mathit{Post}\ s\ (\texttt{read}\ va)\ s'\ \stackrel{\text{def}}{=}\ s\ =\ s'
\end{array}
$$

$$
\begin{array}{l}
\mathit{Pre}\ s\ (\texttt{write}\ va\ val) \stackrel{\text{def}}{=} \mathit{os\_accessible}(va)\ \wedge\ s.\mathit{aos\_activity}\ =\ \mathit{running}\ \wedge \\
\quad \exists\ ma : \mathit{madd},\ \mathit{va\_mapped\_to\_ma}(s, va, ma)\ \wedge \\
\quad \mathit{is\_RW}((s.\mathit{memory}[ma]).\mathit{page\_content}) \\
\mathit{Post}\ s\ (\texttt{write}\ va\ val)\ s'\ \stackrel{\text{def}}{=} \exists\ ma : \mathit{madd},\ \mathit{va\_mapped\_to\_ma}(s, va, ma)\ \wedge \\
\quad s'.\mathit{memory}\ =\ (s.\mathit{memory}[ma := \langle \mathit{RW}(\mathit{Some}\ val), s.\mathit{active\_os}\rangle])\ \wedge \\
\quad s \sim_{\mathit{memory},ma} s'
\end{array}
$$

$$
\begin{array}{l}
\mathit{Pre}\ s\ (\texttt{chmod}) \stackrel{\text{def}}{=} s.\mathit{aos\_activity}\ =\ \mathit{waiting}\ \wedge\ (s.\mathit{oss}[s.\mathit{active\_os}]).\mathit{hcall}\ =\ \mathit{None} \\
\mathit{Post}\ s\ (\texttt{chmod})\ s'\ \stackrel{\text{def}}{=} \\
\quad (\mathit{trusted\_os}(s.\mathit{active\_os})\ \wedge\ s'\ = s.[\mathit{aos\_exec\_mode}, \mathit{aos\_activity}]\mathit{svc}, \mathit{running})\ \vee \\
\quad (\neg\ \mathit{trusted\_os}(s.\mathit{active\_os})\ \wedge\ s'\ = s.[\mathit{aos\_exec\_mode}, \mathit{aos\_activity}]\mathit{usr}, \mathit{running})
\end{array}
$$

$$
\begin{array}{l}
\mathit{Pre}\ s\ (\texttt{page\_pin\_untrusted}\ o\ pa\ t) \stackrel{\text{def}}{=} \\
\quad \neg\ \mathit{trusted\_os}(o)\ \wedge\ s.\mathit{aos\_activity}\ =\ \mathit{waiting}\ \wedge \\
\quad (s.\mathit{oss}[o]).\mathit{hcall}\ =\ \mathit{Some}\ (\mathit{Hyperv\_call\_pin}(pa, t))\ \wedge \\
\quad \mathit{physical\_address\_not\_allocated}(s.\mathit{hypervisor}[o], pa)\ \wedge \\
\quad \exists\ ma : \mathit{madd},\ \mathit{memory\_available}(s.\mathit{memory}, ma) \\
\mathit{Post}\ s\ (\texttt{page\_pin\_untrusted}\ o\ pa\ t)\ s'\ \stackrel{\text{def}}{=} \\
\quad \exists\ ma : \mathit{madd}, \mathit{memory\_available}(s.\mathit{memory}, ma)\ \wedge \\
\quad \mathit{newmem}\ =\ (s.\mathit{memory}[ma := \mathit{newpage}(t, o)])\ \wedge \\
\quad \mathit{newoss}\ =\ (s.\mathit{oss}[o := \langle \mathit{None}, (s.\mathit{oss}[o]).\mathit{curr\_page}\rangle])\ \wedge \\
\quad \mathit{newhyperv}\ =\ (s.\mathit{hypervisor}[o, pa := ma])\ \wedge \\
\quad s'\ = s.[\mathit{oss}, \mathit{hypervisor}, \mathit{memory}]\mathit{newoss}, \mathit{newhyperv}, \mathit{newmem}
\end{array}
$$

Fig. 2. Formal specification of actions semantics

the following properties: i) a trusted OS has no pending hypercalls; ii) if the active OS is in running mode then no hypercall requested by it is pending; iii) if the hypervisor or a trusted OS is running the processor must be in supervisor mode; iv) if an untrusted OS is running the processor must be in user mode; v) the hypervisor maps an OS physical address to a machine address owned by that same OS. This mapping is also injective; vi) all page tables of an OS o map accessible virtual addresses to pages owned by o and not accessible ones to pages owned by the hypervisor; vii) the current page table of any OS is owned by that OS; viii) any machine address ma which is associated to a virtual address in a page table has a corresponding pre-image, which is a physical address, in the hypervisor mapping. All properties have a straightforward interpretation in our model. For example, the first property is captured by the proposition: $\forall\ osi : \mathit{os\_ident},\ \mathit{trusted\_os}(osi) \rightarrow (s.\mathit{oss}[osi]).\mathit{hcall}\ =\ \mathit{None}$. Valid states are invariant under execution, as shall be shown later.

`read` *va*	A guest OS reads virtual address *va*.
`write` *va val*	A guest OS writes value *val* in virtual address *va*.
`new_untrusted` *o va pa*	The hypervisor adds (on behalf of the OS *o*) a new ordered pair (mapping virtual address *va* to the machine address *ma*) to the current memory mapping of the untrusted OS *o*, where *pa* translates to *ma* for *o*.
`del_untrusted` *o va*	The hypervisor deletes (on behalf of the *o* OS) the ordered pair that maps virtual address *va* from the current memory mapping of *o*.
`switch` *o*	The hypervisor sets *o* to be the active OS.
`hcall` *c*	An untrusted OS requires privileged service *c* to be executed by the hypervisor.
`ret_ctrl`	Returns the execution control to the hypervisor.
`chmod`	The hypervisor changes the execution mode from supervisor to user mode, if the active OS is untrusted, and gives to it the execution control.
`page_pin_untrusted` *o pa t*	The memory page that corresponds to physical address *pa* (for untrusted OS *o*) is registered and classified with type *t*.
`page_unpin_untrusted` *o pa*	The memory page that corresponds to physical address *pa* (for the untrusted OS *o*) is un-registered.

Table 1. Actions

3.3 Actions

Table 1 summarises a subset of the actions specified in the model, and their effects. Actions can be classified as follows: i) hypervisor calls `new`, `delete`, `pin`, `unpin` and `lswitch`; ii) change of the active OS by the hypervisor (`switch`); iii) access, from an OS or the hypervisor, to memory pages (`read` and `write`); iv) update of page tables by the hypervisor on demand of an untrusted OS or by a trusted OS directly (`new` and `delete`); v) changes of the execution mode (`chmod`, `ret_ctrl`); and vi) changes in the hypervisor memory mapping (`pin` and `unpin`), which are performed by the hypervisor on demand of an untrusted OS or by a trusted OS directly. These actions model (de)allocation of resources.

Actions Semantics The behaviour of actions is specified by a precondition *Pre* and by a postcondition *Post* of respective types: $Pre : State \rightarrow Action \rightarrow Prop$, and $Post : State \rightarrow Action \rightarrow State \rightarrow Prop$. Figure 2 provides the axiomatic semantics of some relevant actions, namely, `read`, `write`, `chmod` and `page_pin_untrusted` (the names of the auxiliary predicates used should be self-explanatory). Notice that what is specified is the effect the execution of an action has on the state of the platform. In particular, the action `read` does not return the accessed value.

The precondition of the action `read` *va* requires that *va* is accessible by the active OS, that there exists a machine address *ma* to which *va* is mapped, that the active OS is running and that the page indexed by the machine address *ma*

is readable/writable. The postcondition requires the execution of this action to keep the state unchanged. The precondition of the action `write` is identical to that of the action `read`. The postcondition establishes that the state after the execution of the action only differs in the value (*val*) of the page associated to *ma*, which is owned by the active OS. The precondition of the action `chmod` requires that there must not be a pending hypercall for the active OS. The postcondition establishes that after the execution of the action, if the active OS is a trusted one, then the effect on the state is to change its execution mode to supervisor mode. Otherwise, the execution mode is set to user mode. In both cases, the processor mode is set to *running*.

The execution of the action `page_pin_untrusted` requires, in the first place, that the hypervisor is running and that the active OS is untrusted. In addition to that, the OS *o* must be waiting for an hypercall to *pin* the physical address *pa* of type *t*, *pa* must not be already allocated and there must be machine memory available. The effect of the action is to create and allocate at machine address *ma* a new page of type *t* whose owner is the OS *o* and bind, in the hypervisor mapping, the physical address *pa* to *ma*. The rest of the state remains unchanged.

One-step execution The execution of an action is specified by the $\hookrightarrow$ relation:

$$\frac{valid\_state(s) \quad Pre\ s\ a \quad Post\ s\ a\ s'}{s \xhookrightarrow{a} s'}$$

Whenever an action occurs for which the precondition holds, the (valid) state may change in such a way that the action postcondition is established. The notation $s \xhookrightarrow{a} s'$ may be read as *the execution of the action a in a valid state s results in a new state* s'. Note that this definition of execution does not consider the cases where the preconditions of the actions are not fulfilled.

Invariance of valid state One-step execution preserves valid states, that is to say, the state resulting from the execution of an action is also a valid one.

Lemma 1. $\forall\ (s\ s' : State)\ (a : Action),\ s \xhookrightarrow{a} s' \ \rightarrow\ valid\_state(s')$

3.4 Traces

Isolation properties are eventually expressed on execution traces, rather than execution steps; likewise, availability properties are formalized as fairness properties stating that something good will eventually happen in an execution traces. Thus, our formalization includes a definition of execution traces and proof principles to reason about them.

Informally, an execution trace is defined as a stream (an infinite list) of states that are related by the transition relation $\hookrightarrow$, i.e. an object of the form $s_0 \xhookrightarrow{a_0} s_1 \xhookrightarrow{a_1} s_2 \xhookrightarrow{a_2} s_3 \ldots$ such that every execution step $s_i \xhookrightarrow{a_i} s_{i+1}$ is valid. Formally, an execution trace is defined as a stream *ss* of pairs of states and

actions, such that for every $i \geq 0$, $s[i] \xhookrightarrow{a[i]} s[i+1]$, where $ss[i] = \langle s[i], a[i] \rangle$ and $ss[i+1] = \langle s[i+1], a[i+1] \rangle$. We let $Trace$ define the type of traces.

State properties are lifted to properties on pairs of states and actions in the obvious way. Moreover, state properties can be lifted to properties on traces; formally, each predicate P on states can be lifted to predicates $\Box\ P$ (read always P) and $\Diamond\ P$ (read eventually P). The former $\Box\ P$ is defined co-inductively defined by the clause $\Box(P, s :: ss)$ iff $P(s)$ and $\Box(P, ss)$, whereas the latter $\Diamond\ P$ is defined inductively by the clauses $\Diamond(P, s :: ss)$ iff $P(s)$ or $\Diamond(P, ss)$; each modality has an associated reasoning principle attached to its definition. Similar modalities can be defined for relations, and can be used to express isolation properties. In particular, given a relation R on states, and two traces ss_1 and ss_2, we have $\Box(R, ss_1, ss_2)$ iff $R(ss_1[i], ss_2[i])$ for all i.

4 Isolation properties

We formally establish that the hypervisor enforces strong isolation properties: an operating system can only read and modify memory that it owns, and its behavior is independent of the state of other operating systems. The properties are established for a single step of execution, and then extended to traces.

Read isolation Read isolation captures the intuition that no OS can read memory that does not belong to it. Formally, read isolation states that the execution of a `read` va action requires that va is mapped to a machine address ma that belongs to the active OS current memory mapping, and that is owned by the active OS.

Lemma 2. $\forall\ (s\ s' : State)\ (va : vadd)$,
$s \xhookrightarrow{read\ va} s'\ \rightarrow\ \exists\ ma : madd,\ va\_mapped\_to\_ma(s, va, ma)\ \wedge$
$\exists\ pg : page,\ pg = s.memory[ma]\ \wedge\ pg.page\_owned\_by = s.active\_os$

The property is proved by inspection of the pre and postcondition for the `read` action, using the definition of valid state.

Write Isolation Write isolation captures the intuition that an OS cannot modify memory that it does not own. Formally, write isolation states that, unless the hypervisor is running, the execution of any action will at most modify memory pages owned by the active OS or it will allocate a new page for that OS.

Lemma 3. $\forall\ (s\ s' : State)\ (a : Action)\ (ma : madd)$,
$s \xhookrightarrow{a} s'\ \rightarrow\ \neg hyper\_running(s)\ \rightarrow$
$s'.memory[ma] = s.memory[ma] \vee owner\_or\_free(s.memory, ma, s.active\_os)$

where $hyper\_running$ and $owner\_or\_free$ respectively denote that the hypervisor is running, and that the owner of the given machine address is either the given OS or it is free.

The property is proved by case analysis on the action executed. The relevant cases are the actions that are performed by the active OS and that modify the memory; for each such action, the property follows from its pre and postconditions, and from the definition of valid state.

OS Isolation OS isolation captures the intuition that the behavior of any OS does not depend on other OSs states, and is expressed using the notion of *equivalence* w.r.t. an operating system *osi*. Formally, two states s and s' are *osi-equivalent*, denoted $s \equiv_{osi} s'$, if the following conditions are satisfied: i) *osi* has the same hypercall in both states, or no hypercall in both states; ii) the current page tables of *osi* are the same in both states; iii) all page table mappings of *osi* that maps a virtual address to a RW page in one state, must map that address to a page with the same content in the other; iv) the hypervisor mappings of *osi* in both states are such that if a given physical address maps to some RW page, it must map to a page with the same content on the other state. Note that we cannot require that memory contents be the same in both states for them to be *osi*-equivalent, because on a `page_pin` action, the hypervisor can assign an arbitrary (free) machine address to the OS, so we consider *osi*-equivalence without taking into account the actual value of the machine addresses assigned. In particular, two *osi*-equivalent states can have different page table memory pages, which contain mappings from virtual to arbitrary machine addresses, but such that the contents of such an arbitrary machine address be the same on both states, if it corresponds to a RW page. This definition bears some similarity with notions of indistinguishable states used for reasoning about non-interference in object-oriented languages [5].

OS isolation states that *osi*-equivalence is preserved under execution of any action, and is formalized as a "step-consistent" unwinding lemma, see [20].

Lemma 4. $\forall\ (s_1\ s_1'\ s_2\ s_2' : State)\ (a : Action)\ (osi : os\_ident),$
$s_1 \equiv_{osi} s_2\ \rightarrow s_1 \xhookrightarrow{a} s_1'\ \rightarrow\ s_2 \xhookrightarrow{a} s_2'\ \rightarrow\ s_1' \equiv_{osi} s_2'$

The proof of OS isolation relies on write isolation, on Lemma 5, and on an isolation lemma for the case where *osi* is the active OS of both states s_1 and s_2.

The next lemma formalizes a "locally preserves" unwinding lemma in the style of [20], stating that the *osi*-component of a state is not modified when another operating system is executing.

Lemma 5. $\forall\ (s\ s' : State)\ (a : Action)\ (osi : os\_ident),$
$\neg\ os\_action(s, a, osi)\ \rightarrow\ s \xhookrightarrow{a} s'\ \rightarrow\ s \equiv_{osi} s'$

where $os\_action(s, a, osi)$ holds if, in the state s, *osi* is the active and running OS and therefore is executing action a, or otherwise the hypervisor is executing the action a on behalf of *osi*.

Extensions to traces All isolation properties extend to traces, using coinductive reasoning principles. In particular, the extension of OS isolation to traces establishes a non-influence property [18]. Formally, we define for each operating

system *osi* a predicate *same_os_actions* stating that two steps have the same set of actions w.r.t. *osi*: concretely, $same\_os\_actions(osi, ss_1, ss_2)$ holds provided for all i the actions in $ss_1[i]$ and $ss_2[i]$ are the same *os_action* for *osi*, or both are arbitrary actions not related to *osi*.

Lemma 6. $\forall\ (ss_1\ ss_2 : Trace)\ (osi : os\_ident),$
$same\_os\_actions(osi, ss_1, ss_2) \rightarrow (ss_1[0] \equiv_{osi} ss_2[0]) \rightarrow \Box(\equiv_{osi}, ss_1, ss_2)$

For technical reasons related to the treatment of coinductive definitions in Coq (specifically the need for corecursive definitions to be productive), our formalization of non-influence departs from common definitions of non-interference and non-influence, which rely on a purge function that eliminates the actions that are not related to *osi*. One can however define an erasure function *erase* that replaces actions that are not related to *osi* by `silent` actions, and prove for all traces *ss* that $\Box(\equiv_{osi}, ss, erase(osi, ss))$.

5 Availability

An essential property of virtualization platforms is that all guest operating systems are given access to the resources they need to proceed with their execution. In this section, we establish a strong fairness property, showing that if the hypervisor only performs `chmod` actions whenever no hypercall is pending, then no OS blocks indefinitely waiting for its hypercalls to be attended. The assumption on the hypervisor is satisfied by all reasonable implementations of the hypervisor; one possible implementation that would satisfy this restriction is an eager hypervisor which attends hypercalls as soon as it receives them and then chooses an operating system to run next. If this is the case, then when the `chmod` action is executed, no hypercalls are pending on the whole platform.

Formally, the assumption on the hypervisor is modelled by considering a restricted set of execution traces in which the initial state has no hypercall pending, and in `chmod` actions can only be performed whenever no hypercall is pending. Then, the strong fairness property states that: if the hypervisor returns control to guest operating systems infinitely often, then infinitely often there is no pending hypervisor call.

Lemma 7. $\forall\ (ss : Trace), \neg\ hcall(ss[0]) \rightarrow \Box(chmod\_nohcall, ss) \rightarrow$
$\Box(\Diamond\ \neg\ hyper\_running, ss) \rightarrow \Box(\Diamond\ \neg\ hcall, ss)$

where *hcall* and *chmod_nohcall* respectively denote that there is an hypercall pending and that `chmod` actions only arise when no hypercall is pending.

The proof of the strong fairness property proceeds by co-induction and relies on showing that $\neg\ hyper\_running(s) \rightarrow \neg\ hcall(s)$ is an invariant of all traces that satisfy the hypothesis of the lemma.

Note that our strong fairness property is independent of the scheduler: in particular, the hypothesis $\Box(\Diamond\ \neg\ hyper\_running, ss)$ does not guarantee that each operating system will be able to execute infinitely often. Further restricting the implementation of the hypervisor so as to guarantee that the hypervisor is fair to each guest operating system is left for future work.

6 Related work

There have been many efforts to formally verify (parts of) operating systems, see [15] for a survey. The Microsoft Hyper-V verification project focuses on proving the functional correctness of the deployed implementation of the Hyper-V hypervisor [8, 16] or of a simplified, baby, implementation [2]. Using VCC, an automated verifier for annotated C code, these works aim to prove that the hypervisor correctly simulates the execution of the guest operating systems, in the sense that the latter cannot observe any difference from executing on their own on a standard platform. At a more specific level, these works provide a detailed account of many components that are not considered in our work, including page tables [1], devices [3] and cache [8]. The cache is of particular interest from the point of view of security, and Cohen [8] reports on finding cache attacks in the Microsoft Hyper-V verification project. Indeed, the cache constitutes a shared resource which might leak information if not flushed when changing of active operating system. Formalizing the cache and giving sufficient conditions for proving isolation in its presence is a prime goal for future work.

The L4.verified project [14] focuses on proving that the functional correctness of an implementation of seL4, a microkernel whose main application is as an hypervisor running paravirtualized Linux. The implementation consists of approximately 9kLOC of C and 600 lines of assembler, and has been shown to be a valid refinement of a very detailed abstract model that considers for example page tables and I/O devices. Their current work focuses on showing isolation properties; one difference is that in our model the access to a page is restricted to a unique owner, whereas they rely on more flexible capability systems [9].

More recently, the Verve project [23] has initiated the development of a new operating system whose type safety and memory safety has been verified using a combination of type systems and Hoare logic. Outside these projects, several projects have implemented small hypervisors, to reduce the Trusted Computing Base, or with formal verification in mind [22], but we are not aware of any completed proof of functional correctness or security.

Our work is also related to formal verification of isolation properties for separation kernels. Earlier works on separation kernels [13, 12, 17] formalize a simpler model where memory is partitioned a priori. In contrast, our model allows the partition to evolve and comprises three types of addressing modes and is close to those of virtualization platforms, where memory requested by the OSs is dynamicaly allocated from a common memory pool. Dealing with this kind of memory management adds significant complications in isolation proofs.

Our work is also inspired by earlier efforts to prove isolation for smartcard platforms. Andronick, Chetali and Ly [4] use the Coq proof assistant to establish that the JavaCard firewall mechanism ensures isolation properties between contexts—sets of applications that trust each other. Oheimb and coworkers [18, 19] independently verify isolation properties for Infineon SLE 88 using the Isabelle proof assistant. In particular, their work formalizes a notion of non-influence that is closely related to our isolation properties.

7 Conclusion and future work

We have developed an idealized model of a hypervisor and established within this model isolation and availability properties that are expected from virtualization platforms. The formal development is about 20kLOC of Coq (see Figure 3), including proofs, and forms a suitable basis for reasoning about hypervisors.

Model and basic lemmas	4.8k
Valid state invariance	8.0k
Read and write isolation	0.6k
OS Isolation and lemmas	6.0k
Traces, safety and availability	1.0k
Total	20.4k

Fig. 3. LOC of Coq development

There are several directions for future work: one immediate direction is to complete our formalization with a proof of correctness of the hypervisor, as in the Hyper-V verification project. We also intend to enrich our model with shared resources; concretely, we intend to concentrate on the cache and to provide sufficient conditions for isolation properties to hold in its presence. Another immediate direction is to prove isolation and availability properties on an implementation of the hypervisor, using recent work by the authors. Finally, it is of interest to understand how to adapt our models to other virtualization paradigms such as full virtualization and microvisors.

Acknowledgments Thanks to Andrés Krapf, Anne Pacalet, Francois Armand and Christian Jacquemot for their involvement at early stages of the project, Julio Pérez for his contribution on proof checking and June Andronick, Gerwin Klein, Toby Murray, and FM reviewers for feedback on the paper.

References

1. E. Alkassar, E. Cohen, M. Hillebrand, M. Kovalev, and W. Paul. Verifying shadow page table algorithms. In R. Bloem and N. Sharygina, editors, *Formal Methods in Computer-Aided Design, 10th International Conference (FMCAD'10)*, Switzerland, 2010. IEEE CS.
2. E. Alkassar, M. Hillebrand, W. Paul, and E. Petrova. Automated verification of a small hypervisor. In G. Leavens, P. O'Hearn, and S. Rajamani, editors, *Verified Software: Theories, Tools, Experiments*, volume 6217 of *LNCS*, pages 40–54. Springer, 2010.
3. E. Alkassar, W. Paul, A. Starostin, and A. Tsyban. Pervasive verification of an os microkernel: Inline assembly, memory consumption, concurrent devices. In *Third International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'10)*, volume 6217 of *LNCS*, pages 71–85, Edinburgh, 2010. Springer.
4. J. Andronick, B. Chetali, and O. Ly. Using Coq to Verify Java Card Applet Isolation Properties. In D. A. Basin and B. Wolff, editors, *International Conference on Theorem Proving in Higher Order Logics (TPHOLs'03)*, volume 2758 of *LNCS*, pages 335–351. Springer-Verlag, 2003.
5. A. Banerjee and D. Naumann. Stack-based access control for secure information flow. *Journal of Functional Programming*, 15:131–177, March 2005. Special Issue on Language-Based Security.

6. P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.
7. M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.
8. E. Cohen. Validating the microsoft hypervisor. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM '06*, volume 4085 of *LNCS*, pages 81–81. Springer, 2006.
9. D. Elkaduwe, G. Klein, and K. Elphinstone. Verified protection model of the sel4 microkernel. In *Proceedings of the 2nd international conference on Verified Software: Theories, Tools, Experiments*, VSTTE '08, pages 99–114. Springer, 2008.
10. T. Garfinkel and A. Warfield. What virtualization can do for security. *;login: The USENIX Magazine*, 32, December 2007.
11. Robert P. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7:34–45, June 1974.
12. David Greve, Matthew Wilding, and W. Mark Van Eet. A separation kernel formal security policy. In *Proc. Fourth International Workshop on the ACL2 Theorem Prover and Its Applications*, 2003.
13. Constance L. Heitmeyer, Myla Archer, Elizabeth I. Leonard, and John McLean. Formal specification and verification of data separation in a separation kernel for an embedded system. In *Proceedings of the 13th ACM conference on Computer and communications security*, CCS '06, pages 346–355, NY, USA, 2006. ACM.
14. G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. *Communications of the ACM (CACM)*, 53(6):107–115, June 2010.
15. Gerwin Klein. Operating system verification – an overview. *Sādhanā*, 34(1):27–69, February 2009.
16. D. Leinenbach and T. Santen. Verifying the microsoft hyper-v hypervisor with vcc. In A. Cavalcanti and D. Dams, editors, *FM 2009*, volume 5850 of *LNCS*, pages 806–809. Springer, 2009.
17. W. Martin, P. White, F.S. Taylor, and A. Goldberg. Formal construction of the mathematically analyzed separation kernel. In *The Fifteenth IEEE International Conference on Automated Software Engineering*, 2000.
18. David von Oheimb. Information flow control revisited: Noninfluence = Noninterference + Nonleakage. In P. Samarati, P. Ryan, D. Gollmann, and R. Molva, editors, *Computer Security – ESORICS 2004*, volume 3193 of *LNCS*, pages 225–243. Springer, 2004.
19. David von Oheimb, Volkmar Lotz, and Georg Walter. Analyzing SLE 88 memory management security using Interacting State Machines. *International Journal of Information Security*, 4(3):155–171, 2005.
20. J. M. Rushby. Noninterference, Transitivity, and Channel-Control Security Policies. Technical Report CSL-92-02, SRI International, 1992.
21. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.2*, 2008.
22. H. Tews, Tjark Weber, E. Poll, and M.C.J.D. van Eekelen. Formal Nova interface specification. Technical Report ICIS–R08011, Radboud University Nijmegen, May 2008. Robin deliverable D12.
23. Jean Yang and Chris Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In *Proceedings of PLDI'10*, pages 99–110. ACM, 2010.