Formally Verified Implementation of an Idealized Model of Virtualization

Gilles Barthe[1], Gustavo Betarte[2], Juan Diego Campo[2], Jesús Mauricio Chimento[3], and Carlos Luna[2]

1 **IMDEA Software, Madrid, Spain**
gilles.barthe@imdea.org
2 **InCo, Facultad de Ingeniería, Universidad de la República, Uruguay**
{gustun,jdcampo,cluna}@fing.edu.uy
3 **FCEIA, Universidad Nacional de Rosario, Argentina**
checholcc@gmail.com

Abstract

VirtualCert is a machine-checked model of virtualization that can be used to reason about isolation between operating systems in presence of cache-based side-channels. In contrast to most prominent projects on operating systems verification, where such guarantees are proved directly on concrete implementations of hypervisors, VirtualCert abstracts away most implementations issues and specifies the effects of hypervisor actions axiomatically, in terms of preconditions and postconditions. Unfortunately, seemingly innocuous implementation issues are often relevant for security. Incorporating the treatment of errors into VirtualCert is therefore an important step towards strengthening the isolation theorems proved in earlier work. In this paper, we extend our earlier model with errors, and prove that isolation theorems still apply. In addition, we provide an executable specification of the hypervisor, and prove that it correctly implements the axiomatic model. The executable specification constitutes a first step towards a more realistic implementation of a hypervisor, and provides a useful tool for validating the axiomatic semantics developed in previous work.

1998 ACM Subject Classification D.2.4 Software/Program Verification, D.4.6 Security and Protection

Keywords and phrases virtualization, cache and TLB, executable specification, error management, isolation

Digital Object Identifier 10.4230/LIPIcs.TYPES.2013.45

1 Introduction

Virtualization is a prominent technology that allows high-integrity, safety-critical, systems and untrusted, non-critical, systems to coexist securely on the same platform and efficiently share its resources. To achieve the strong security guarantees requested by these application scenarios, virtualization platforms impose a strict control on the interactions between their guest systems. While this control theoretically guarantees isolation between guest systems, implementation errors and side-channels often lead to breaches of confidentiality, allowing a malicious guest system to obtain secret information, such as a cryptographic key, about another guest system.

Over the last few years, there have been significant efforts to prove that virtualization platforms deliver the expected, strong, isolation properties between operating systems. The most prominent efforts in this direction are within the Hyper-V [13, 19] and L4.verified [17]

projects, which aim to derive strong guarantees for concrete implementations: more specifically, Murray *et al.* [24] recently presented a machine-checked information flow security proof for the seL4 microkernel.

Earlier work

In [4, 5], we have pursued a complementary approach in which verification of isolation properties is conducted in an idealized model of virtualization, named VirtualCert [28]. In comparison with the Hyper-V and L4.verified projects, our proofs are based on an axiomatization of the semantics of a hypervisor, and abstract away many details from the implementation; on the other hand, our model integrates caches and Translation Lookaside Buffers (TLBs), two security relevant components that are not considered in these works. Specifically, we formalize using the Coq proof assistant [31] the semantics of a hypervisor. The semantics accounts for cache-based side-channels, by allowing that a malicious operating system can draw observations from the history of the cache; the treatment of cache-based side-channels is inspired from earlier work on physically observable cryptography [23], but is specialized to caches and TLBs. Then, we prove that, for a wide range of replacement and write policies, flushing the cache upon switching between guest operating systems ensures OS isolation and prevents access-driven cache-based attacks [34].

Contributions

The axiomatic semantics of [4, 5] only considers correct execution. The first contribution of this paper is an implementation of a hypervisor in the programming language of Coq, and a proof that it realizes the axiomatic semantics. Although it remains idealized and far from a realistic hypervisor, the implementation arguably provides a useful mechanism for validating the axiomatic semantics.

The implementation is total, in the sense that it computes for every state and action a new state or an error. Thus, soundness is proved with respect to an extended axiomatic semantics in which transitions may lead to errors. The second contribution of this paper is a proof that OS isolation remains valid for executions that may trigger errors.

Formal language and notation used

The Coq proof assistant [31, 9] is a free open source software that provides a (dependently typed) functional programming language and a reasoning framework based on higher order logic to perform proofs of programs. As examples of its applicability, Coq has been used as a framework for formalizing programming environments and designing special platforms for software verification: the Gemalto and Trusted Logic companies obtained the level CC EAL 7 of certification for their formalization, developed in Coq, of the security properties of the JavaCard platform [11, 10, 1]; Leroy and others developed in Coq a certified optimizing compiler for a large subset of the C programming language [20]; Barthe and others used Coq to develop Certicrypt, an environment of formal proofs for computational cryptography [7].

We developed our specification in the Calculus of Inductive Constructions (CIC) [14, 15, 27] – formal language that combines a higher-order logic and a richly-typed functional programming language – using Coq.

We freely use enumerated types, option types, lists, streams and records. Enumerated types and (parametric) sum types are defined using Haskell-like notation; for example, we define for every type T the type $\mathit{option}\ T \stackrel{\text{def}}{=} \mathit{None} \mid \mathit{Some}\ (t : T)$. 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

and field update are respectively written as $r.l$ and $r'[l := v]$; we also use simultaneous field update, which is defined in the usual way. We make an extensive use of partial maps, and bounded partial maps: the type of partial maps from objects of type A into objects of type B is written $A \mapsto B$, and the type of partial maps from A to B whose domain is of size smaller or equal to k (where k is a natural number) is written as $A \mapsto_k B$. Application of a map m on an object a of type A is denoted $m[a]$ and map update is written $m[a := b]$, where b overwrites the value, if any, associated to the key a.

Organization of the paper

The rest of the paper is organized as follows. Section 2 provides a brief account of the basic components of the idealized model focusing on the memory model and the notion of state that has been formalized. Section 3 describes the formal axiomatic and executable semantics of the hypervisor and outlines the proof of correctness of the implementation. In section 4 we present the isolation theorems for the model extended with execution errors. Section 5 discusses related work and concludes.

The formal development can be found in [28], and can be verified using Coq.

2 Background

In this section we provide insights into the basic structures of VirtualCert, namely, the memory model and the set of (valid) states.

Memory model

The formalized memory model includes the main memory of the platform, various kinds of memory spaces, and the cache and the TLB. Our modelling choices are guided by Xen [3], and specifically, by Xen on ARM [16]. As shown in Figure 1 there are three different types of memory addresses: i) the machine addresses (written *madd*) model real hardware memory on the host machine and it is never directly accessed by the guest operating systems, ii) the physical addresses (*padd*) are an abstraction provided by the hypervisor, in order for the guest operating systems to use a contiguous memory space when dealing with its memory pages. The mapping between physical and machine addresses is managed exclusively by the hypervisor, and is transparent to the guest operating systems, and iii) the virtual addresses (*vadd*) are used by applications running on guest operating systems. Each OS has a designated portion of its virtual address space that is reserved for the hypervisor to attend hypercalls. 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. The hypervisor maintains page tables that map virtual addresses to machine addresses in special memory pages. The operating systems must call the hypervisor to modify these mappings.

The figure also shows the cache and the TLB. The cache is indexed by a virtual address, modeling a Virtually Indexed Virtually Tagged (VIVT) cache, and holds a (partial) copy of memory pages. The TLB is used in conjunction with the current page table of the active OS to map virtual to machine addresses. In [5], we present a brief overview of cache management, where we describe different cache types and alternatives policies for implementing cache content management, in particular concerning the update and replacement of cache information.

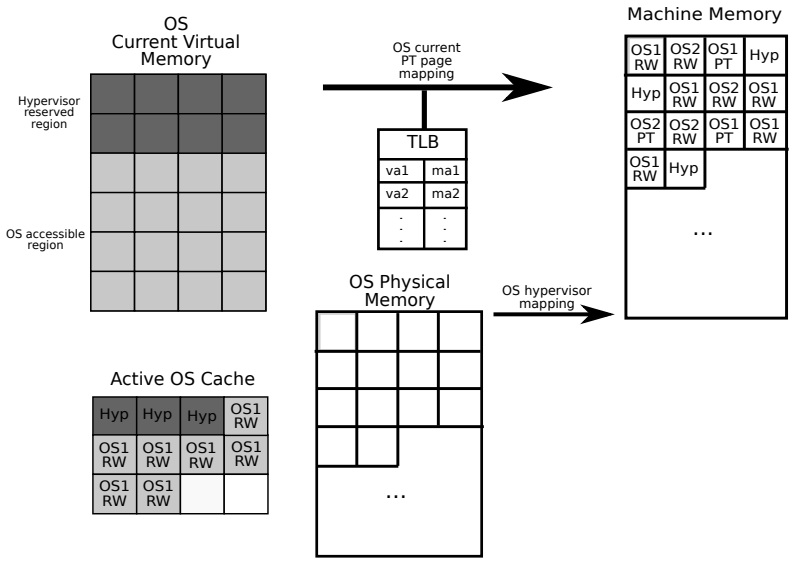

Figure 1 Memory model of the platform.

Platform states

States are modeled as records:

$$
\begin{aligned}
State \stackrel{\text{def}}{=} \{ \quad & oss && : oss\_map, \\
& active\_os && : os\_ident, \\
& mode && : exec\_mode, \\
& activity && : os\_activity, \\
& hypervisor && : hypervisor\_map, \\
& memory && : machine\_memory, \\
& cache && : cache\_vivt, \\
& tlb && : tlb\_struct \} .
\end{aligned}
$$

We define a type *os_ident* of identifiers for guest OSs and a predicate *trusted_os* that separates between trusted and untrusted OSs. The state contains information about each guest OS such as its current page table, and whether the OS has a pending hypercall to be resolved. Formally this information is captured by a mapping *oss_map* that associates OS identifiers with objects of type *os*, where

$$
\begin{aligned}
& os \stackrel{\text{def}}{=} \{curr\_page : padd, hcall : option\ Hyper\_call\} , \\
& oss\_map \stackrel{\text{def}}{=} os\_ident \ \mapsto \ os .
\end{aligned}
$$

The state also stores the current active operating system, and the execution mode of the CPU (*user* or *supervisor* mode). Guest operating systems execute in *user* mode (where some privileged instructions are not available) and the hypervisor executes in *supervisor* mode. The activity registers whether the active OS is currently *running* or *waiting* for a hypercall to be resolved. 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. A memory page consists of a page content (either a readable/writable value, an OS

page table mapping, or nothing) and a reference to the page owner (the hypervisor, an OS, or none). Formally:

$$
\begin{aligned}
&\mathit{machine\_memory} \stackrel{\text{def}}{=} \mathit{madd} \mapsto \mathit{page}\,,\\
&\mathit{content} \stackrel{\text{def}}{=} \mathit{RW}\ (v : \mathit{option}\ \mathit{Value}) \mid \mathit{PT}\ (\mathit{va\_to\_ma} : \mathit{vadd} \mapsto \mathit{madd}) \mid \mathit{Other}\,,\\
&\mathit{page\_owner} \stackrel{\text{def}}{=} \mathit{Hyp} \mid \mathit{Os}\ (\mathit{osi} : \mathit{os\_ident}) \mid \mathit{No\_Owner}\,,\\
&\mathit{page} \stackrel{\text{def}}{=} \{\ \mathit{page\_content} : \mathit{content}, \mathit{page\_owned\_by} : \mathit{page\_owner}\ \}\,.
\end{aligned}
$$

Finally, the cache and the TLB of the platform are formalized as partial maps, whose domains are bounded in size with positive fixed constants *size_cache* and *size_tlb*:

$$
\begin{aligned}
&\mathit{cache\_vivt} \stackrel{\text{def}}{=} \mathit{vadd} \mapsto_{\mathit{size\_cache}} \mathit{page}\,,\\
&\mathit{tlb\_struct} \stackrel{\text{def}}{=} \mathit{vadd} \mapsto_{\mathit{size\_tlb}} \mathit{madd}\,.
\end{aligned}
$$

We define a notion of valid state, through the predicate *valid_state* on states, that captures essential properties of the platform. The definition is provided in Appendix A.1.

3 Verified implementation

In this section we first provide a short account of the axiomatic semantics of the hypervisor, to proceed to motivate the extension of the model with execution errors. Then we describe the executable specification and show that it constitutes a correct implementation of the behavior specified by the idealized model.

3.1 Actions semantics

The axiomatic semantics of the hypervisor is modeled by defining a set of actions, and providing their semantics as state transformers. Table 1 summarises a small subset of the actions specified in our model. The complete set of actions is included in Appendix A.2. Actions can be classified as follows:

- hypervisor calls `new`, `delete`, `pin`, `unpin` and `lswitch`;
- change of the active OS by the hypervisor (`switch`);
- access, from an OS or the hypervisor, to memory pages (`read` and `write`);
- update of page tables by the hypervisor on demand of an untrusted OS or by a trusted OS directly (`new` and `delete`);
- changes of the execution mode (`chmod`, `ret_ctrl`); and
- 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.

The behaviour of actions is specified by a precondition *Pre* and by a postcondition *Post* of respective types:

$$
\begin{aligned}
\mathit{Pre} &: \mathit{State} \to \mathit{Action} \to \mathit{Prop}\,,\\
\mathit{Post} &: \mathit{State} \to \mathit{Action} \to \mathit{State} \to \mathit{Prop}\,.
\end{aligned}
$$

Figure 2 provides the axiomatic semantics of the `write` action.

The precondition of the action `write` *va val* says that there exists a machine address *ma* such that *va* is associated to it (*va_mapped_to_ma*) and that the page associated to it in the memory is readable/writable (*is_RW*); that the guest OS activity must be running; and

Table 1 Actions.

`read_hyper` *va*	The hypervisor reads virtual address *va*.
`write` *va val*	A guest OS writes value *val* in virtual address *va*.
`new_tr` *va pa*	The virtual address *va* is mapped to the machine address *ma* in the memory mapping of the trusted active OS, where *pa* translates to *ma* for the active OS.
`switch` *o*	The hypervisor sets *o* to be the active OS.
`lswitch_untr` *o pa*	The hypervisor changes the current memory mapping of the untrusted active OS, to be the one located at physical address *pa*.
`hcall` *c*	An untrusted OS requires privileged service *c* to be executed by the hypervisor.
`pin_untr` *o pa t*	The memory page that corresponds to physical address *pa* (for untrusted OS *o*) is registered and classified with type *t*.
`unpin_untr` *o pa*	The memory page that corresponds to physical address *pa* (for the untrusted OS *o*) is un-registered.

$$
\begin{array}{l}
Pre\ s\ (\texttt{write}\ va\ val) \stackrel{\text{def}}{=} \exists\ (ma : madd),\\
\quad va\_mapped\_to\_ma(s, va, ma)\ \wedge\ is\_RW(s.memory[ma].page\_content)\ \wedge\\
\quad os\_accessible(va)\ \wedge\ s.activity = running\\
\\
Post\ s\ (\texttt{write}\ va\ val)\ s' \stackrel{\text{def}}{=} \exists\ (ma : madd)\ (pg : page),\\
\quad \texttt{let}\ new\_pg\ :=\ \{RW(Some\ val), pg.page\_owned\_by\}\ \texttt{in}\\
\quad va\_mapped\_to\_pg\_cache(s, va, pg)\ \wedge\ va\_mapped\_to\_ma\_cache(s, va, ma)\ \wedge\\
\quad s' = s \left[\begin{array}{rcl} mem & := & (s.memory[ma := new\_pg]),\\ cache & := & cache\_add(fix\_cache\_synonym(s.cache, ma), va, new\_pg),\\ tlb & := & tlb\_add(s.tlb, va, ma)\end{array}\right]
\end{array}
$$

Figure 2 Axiomatic specification of action `write`.

that *va* must be accessible by the active guest OS (*os_accessible*). Its postcondition sets up that the only variations in the state after executing this action can be produced in the value of the page associated to *ma* in memory, and in the values stored in the cache and the TLB. It is not hard to see that, as the cache uses a write-through policy, both the memory and the cache are updated when a write is performed. As explained in [5], a cache c_2 is the result of updating a cache c_1 with a pair *va* and *pg*, written $c_2 = cache\_add(c_1, va, pg)$, iff

$$
\begin{array}{l}
pg\ =\ c_2[va]\ \wedge\\
\forall\ (va' : vadd)\ (pg' : page),\ va\ \neq\ va'\ \rightarrow\ pg'\ =\ c_2[va']\ \rightarrow\ pg'\ =\ c_1[va']\,.
\end{array}
$$

The definition of $c_2 = tlb\_add(c_1, va, ma)$ is analogous. Moreover, in order to avoid aliasing problems we fix synonyms before adding a new entry into the cache using the function *fix_cache_synonym*. The result of $fix\_cache\_synonym(c_1, ma)$ is a cache c_2 whose indexes (virtual addresses) are translated to machine addresses ma' which differ from ma. We recall that we are modeling a VIVT cache.

3.2 Error management

There can be attempts to execute an action on a state that does not verify the precondition of that action. In the presence of one such situation the system answers with a corresponding error code. These error codes are defined in our model by the enumerated type *ErrorCode*.

Table 2 Preconditions and error codes.

Action	*Failure*	*Error Code*
`write` *va val*	$s.aos\_activity \neq running$	$wrong\_os\_activity$
	$\neg\ va\_mapped\_to\_ma(s, va, ma)$	$invalid\_vadd$
	$\neg\ os\_accessible(va)$	$no\_access\_va\_os$
	$\neg\ is\_RW(s.memory[ma].page\_content)$	$wrong\_page\_type$
`new_tr` *va pa*	$s.aos\_activity \neq running$	$wrong\_os\_activity$
	$\neg\ os\_accessible(va)$	$no\_access\_va\_os$
	$\neg\ trusted\_os(osi)$	$os\_trust\_failure$
	$\neg\ page\_of\_OS(s.active\_os, pa, ma)$	$wrong\_owner$
`lswitch_untr` *osi pa*	$s.aos\_activity \neq waiting$	$wrong\_os\_activity$
	$trusted\_os(osi)$	$os\_trust\_failure$
	$\neg\ is\_PT(s.memory[ma].page\_content)$	$wrong\_page\_type$
	$\neg\ lswitch\_hypercall(s.oss[osi].hcall)$	$wrong\_pending\_hcall$
	$\neg\ page\_of\_OS(s.activeos, pa, ma)$	$wrong\_owner$
`unpin_untr` *osi pa*	$s.aos\_activity \neq waiting$	$wrong\_os\_activity$
	$trusted\_os(osi)$	$os\_trust\_failure$
	$\neg\ page\_unpin\_hypercall(s.oss[osi].hcall)$	$wrong\_pending\_hcall$
	$\neg\ pa\_not\_curr\_page(s, s.oss, pa)$	$wrong\_currpage\_add$
	$s.hypervisor[osi][pa] \neq\ ma$	$invalid\_madd$
	$\neg\ no\_va\_mapped\_to\_ma(s, osi, ma)$	$invalid\_vadd$

We define the relation between an error code and the unfulfilled precondition of an action with the predicate *ErrorMsg*. Formally,

$$ErrorMsg : State \to Action \to ErrorCode \to Prop$$

where *ErrorMsg* s a ec means that the execution of the action a in the state s generates the error ec. In Table 2 we show some examples about error codes associated to unverified preconditions of some actions of our model. Notice that in the case of the `write` action, for instance, to each of the propositions that compose the precondition of that action there corresponds an element of *ErrorCode* that indicates the failure of the state s to satisfy that proposition.

Executions with error management

Executing an action a over a state s produces a new state s' and a corresponding answer r (denoted $s \xrightarrow{a/r} s'$), where the relation between the former state and the new one is given by the postcondition relation *Post*.

$$\frac{valid\_state(s) \qquad Pre(s, a) \qquad Post(s, a, s')}{s \xrightarrow{a/ok} s'}$$

$$\frac{valid\_state(s) \qquad ErrorMsg(s, a, ec)}{s \xrightarrow{a/error\ \ ec} s}$$

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 $s \xrightarrow{a/ok} s'$ may be read

as *the execution of the action a in a valid state s results in a new state s′*. However, if the precondition is not satisfied, then the state s remains unchanged and the system answer is the error message determined by the relation $ErrorMsg$.

Formally, the possible answers of the system are defined by the following type:

$$Response \stackrel{\text{def}}{=} ok : Response \mid error : ErrorCode \rightarrow Response$$

where *ok* is the answer resulting from a successful execution of an action.

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

▶ **Lemma 1** (Validity is invariant).
$\forall\ (s\ s' : State)(a : Action)(r : Response),$
$valid\_state(s)\ \rightarrow s \xrightarrow{a/r} s' \rightarrow valid\_state(s')\ .$

Platform state invariants, such as state validity, are useful to analyze other relevant properties of the model. In particular, the results presented in this work are obtained from valid states of the platform.

3.3 Executable specification

The executable specification of the hypervisor has been written using the Coq proof assistant and it ultimately amounts to the definition of functions that implement action execution. The functions have been defined so as to conform to the axiomatic specification of action execution as provided by the idealized model. The implementation of the hypervisor consists of a set of `Coq` functions, such that for every predicate involved in the axiomatic specification of action execution there exists a function which stands for the functional counterpart of that predicate. An important characteristics of our formalization is that the definition of state that is used for defining the executable semantics of the hypervisor is exactly the same as the one introduced in the idealized model. This simplifies the formal proof of soundness between the inductive and the functional semantics of the hypervisor. The execution of the virtualization platform consists of a (potentially infinite) sequence of action executions starting in an (initial) platform state. The output of the execution is the corresponding sequence of memory states (the trace of execution) obtained while executing the sequence of actions.

3.3.1 Action execution

The execution of actions has been implemented as a *step* function, that given a memory state s and an action a invokes the function that implements the execution of a in s, which in turn returns an object res of type *Result*:

$$Result \stackrel{\text{def}}{=} \{\ resp\ :\ Response, st\ :\ State\ \}$$

where *res.resp* is either an error code *ec*, if the precondition of the actions does not hold in state s, or otherwise the value *ok*, and the state *res.st* represents the execution effect. The *step* function acts basically as an action dispatcher. Figure 3, which shows the structure of the dispatcher, details the branch corresponding to the dispatching of action `write`, which is the action we shall use along this section to illustrate the working of the implementation.

The functions invoked in the branches, like *write_safe*, are state transformers whose definition follows this pattern: first it is checked whether the precondition of the action is

```
Definition step s a :=
  match a with
        | ... ⇒ ...
        | Write va val ⇒ write_safe(s, va, val)
        | ... ⇒ ...
  end.
```

Figure 3 The *step* function.

```
Definition write_safe (s : state) (va : vadd) (val : value) : Result :=
  match write_pre(s, va, val) with
        | Some ec ⇒ {error(ec), s}
        | None ⇒ {ok, write_post(s, va, val)}
  end.
```

Figure 4 Execution of `write` action.

```
Definition write_pre (s : state) (va : vadd) (val : value) : option ErrorCode :=
  match get_os_ma(s, va) with
    | None ⇒ Some invalid_vadd
    | Some ma
      ⇒ match page_type(s.memory, ma) with
        | Some RW
          ⇒ match aos_activity(s) with
            | Waiting ⇒ Some wrong_os_activity
            | Running
              ⇒ if vadd_accessible(s, va)
                then None
                else Some no_access_va_os
            end
        | _ ⇒ Some wrong_page_type
        end
  end.
```

Figure 5 Validation of `write` action precondition.

satisfied in state s, and then, if that is the case, the function that implements the execution of the action is invoked, otherwise, the state s, unchanged, is returned along with an appropriate response.

In Figure 4 we show the definition of the function that implements the execution of the `write` action. The Coq code of this function, together with that of the remaining functions, can be found in [28].

The function *write_pre* is defined as the nested validation of each of the properties of the precondition (see Figure 5). The function *write_post*, shown in Figure 6, implements the expected behavior of the `write` action: when a new value has to be written in a certain virtual address *va*, first it must be checked whether *va* is in the cache (i.e. is an index of the cache). If that is the case, then the function updates both the cache and the memory,

```
Definition write_post (s : state) (va : vadd) (val : value) : state :=
  match s.cache[va] with
  | Value old_pg ⇒
    let new_pg := Page (RW_c (Some val)) (page_owned_by old_pg) in
    let val_ma := va_mapped_to_ma_system(s, va) in
    match val_ma with
    | Value ma ⇒
      s · [ mem := s.memory[ma := new_pg],
            cache := fcache_add(fix_cache_synonym(s.cache, ma), va, new_pg) ]
    | Error _ ⇒ s end
  | Error _ ⇒
    match s.tlb[va] with
    | Value ma ⇒
      match s.memory[ma] with
      | Value old_pg ⇒
        let new_pg := Page (RW_c (Some val)) (page_owned_by old_pg) in
        s · [ mem := s.memory[ma := new_pg],
              cache := fcache_add(fix_cache_synonym(s.cache, ma), va, new_pg) ]
      | Error _ ⇒ s end
    | Error _ ⇒
      match va_mapped_to_ma_currentPT(s, va) with
      | Value ma ⇒
        match s.memory[ma] with
        | Value old_pg ⇒
          let new_pg := Page (RW_c (Some val)) (page_owned_by old_pg) in
          s · [ mem := s.memory[ma := new_pg],
                cache := fcache_add(fix_cache_synonym(s.cache, ma), va, new_pg),
                tlb := ftlb_add(s.tlb, va, ma) ]
        | Error _ ⇒ s end
      | Error _ ⇒ s end end end.
```

Figure 6 Effect of `write` execution

because it implements a write-through policy. Otherwise, i.e. if the virtual address va is not already in the cache, the machine address associated to va has to be determined in order to write the new value in memory. First, the TLB is inspected to check whether va has already been translated. If there is a translation of va in the TLB, then the machine address is used to update the memory and the new entry $\langle va,\ new\_pg\rangle$ is added to the cache. If there is no translation of va in the TLB, then the corresponding machine address has to be recovered using the current page table of the active guest OS. Once that translation has been found, the memory is updated, the new entry $\langle va,\ new\_pg\rangle$ is added to the cache and the corresponding translation of va is added to the TLB.

3.3.2 Cache and TLB update

In the axiomatic semantics of cache and TLB management the replacement policy has been left abstract. For the execution semantics we have chosen to implement a simple FIFO replacement mechanism. However, this behavior is encapsulated in the definition

```
Definition fcache_add (c : cache_struct) (va : vadd) (pg : page) : cache_struct :=
  if map_valid_index(c, va)
  then map_add(c, va, pg)
  else if is_full_cache(c)
       then fifo_replace(c, va, pg)
       else fifo_add(c, va, pg).
```

Figure 7 Cache update.

of the functions *fcache_add* and *ftlb_add*, which implement cache and TLB replacement, respectively. Therefore, for the implementation of an alternative replacement policy it suffices to modify correspondingly these two functions leaving the rest of the code unchanged. Figure 7 shows the definition of the *fcache_add* function: first, it is checked whether the virtual address *va* is the index of an entry of the cache *c* (*map_valid_index*). If this is the case, it suffices to perform a simple update of *c* with the page *pg* (caches are implemented as bounded maps of virtual addresses to machine addresses). Otherwise, the behaviour of the function depends on whether *c* has room for a new entry or it is full (*is_full_cache*). If *c* is full, the cache update, and entry eviction, is handled using the FIFO replacement algorithm (*fifo_replace*). If there is room left for a new entry, then *c* must be updated following the FIFO replacement algorithm guidelines for adding new entries in the cache (*fifo_add*). The definitions of the replacement and update function for the TLB are analogous.

3.4 Soundness

We proceed now to outline the proof that the executable specification of the hypervisor correctly implements the axiomatic model. It has been formally stated as a soundness theorem and verified using the Coq proof assistant.

▶ **Theorem 2** (Soundness of hypervisor implementation).
$\forall\ (s : State)\ (a : Action),$
$valid\_state(s) \rightarrow\ s \xrightarrow{a/step(s,a).resp} step(s,a).st\,.$

The proof of this theorem follows by, in the first place, performing a case analysis on $Pre(s,a)$ (this predicate is decidable) and then: if $Pre(s,a)$ applying Lemma 3; otherwise applying Lemma 5.

▶ **Lemma 3** (Soundness of valid execution).
$\forall\ (s : State)\ (a : Action),$
$valid\_state(s) \rightarrow\ Pre(s,a)\ \rightarrow$
$s \xrightarrow{a/ok} step(s,a).st\ \wedge\ step(s,a).resp = ok\,.$

The proof of Lemma 3 proceeds by applying functional induction on $step(s,a)$ and then by providing the corresponding proof of soundness of the function that implements the execution of each action. Thus, in the case of the action `write` we have stated and proved Lemma 4. This lemma, in turn, follows by performing a case analysis on the result of applying the function *write_pre* on *s* and the action: if the result is an error code then the thesis follows by contradiction. Otherwise, it follows by the correctness of the function *write_post*.

▶ **Lemma 4** (Correctness of `write` execution).
$\forall\ (s : State)\ (va : vadd)\ (val : value),$
$valid\_state(s) \rightarrow\ Pre(s, (\texttt{write}\ va\ val)) \rightarrow$
$Post(s, (\texttt{write}\ va\ val), write\_post(s, va, val))\,.$

As to Lemma 5, the proof also proceeds by first applying functional induction on $step(s, a)$. Then, for each action a, it is shown that if $\neg Pre(s, a)$ the execution of the function that implements that action yields the values returned by the branch corresponding to the case that the function that validates the precondition of the action a in state s fails, i.e., an error code ec and the (unchanged) state s.

▶ **Lemma 5** (Soundness of error execution).
$\forall\ (s : State)\ (a : Action),$
$valid\_state(s) \rightarrow\ \neg Pre(s, a) \rightarrow\ \exists\ (ec : ErrorCode),$
$step(s, a).st = s\ \wedge\ step(s, a).resp = error(ec)\ \wedge\ ErrorMsg(s, a, ec)\,.$

4 Isolation

Isolation theorems ensure that the virtualization platform protects guest operating systems against each other, in the sense that a malicious operating system cannot gain information about another victim operating system executing on the same platform. In earlier work [5], we adopted ideas from physical cryptography and in particular the idea of leakage function to model possible leaks of information via the cache, and prove that the virtualization platform can guarantee perfect isolation by flushing the cache at every context switch. In this section, we extend the proof of OS isolation from [5], yielding modifications in some key technical definitions and lemmas below, so that it accounts for errors in execution traces.

4.1 OS Isolation

OS isolation is a 2-safety property [32, 12], cast in terms of two executions of the system, and is closely related to the non-influence property studied by Oheimb and co-workers [25, 26]. Unfortunately, the technology for verifying 2-safety properties is not fully mature, making their formal verification on large and complex programs exceedingly challenging.

Informally, OS isolation states that starting from states with the same information for an operating system *osi*, *osi* cannot distinguish between the two traces, as long as it executes the same actions in both. This captures the idea that the execution of *osi* does not depend on the state or behaviour of the other systems, even in the presence of erroneous executions.

Note that there is one particular error (the *out_of_memory* error in [28]) that can in principle influence the execution of an operating system, if during its execution the platform runs out of memory. Since we are specifically interested in modelling observations on states (and the cache, in particular), we treat this error as transparent for the executing operating system, and only make sure it does not modify the state. This is consistent with what usually happens in real implementations, where there are no data leaks from the victims when the platform runs out of memory, and the only information an attacker learns is the total memory consumption of the other operating systems in the platform. Additionally it is possible, in this case, to assign to each guest OS a fixed pool of memory from which to allocate, so whether allocation succeeds or fails for one OS doesn't depend on what any other guest OS does.

To formalize OS isolation we use a notion of state equivalence w.r.t. an operating system *osi*. The definition of *osi-equivalence* ($\equiv_{osi}$), which is stated in Appendix A.3, coincides

with the one used in [4]; in particular, it does not mention the cache and the TLB. However, one can prove that it entails some form of cache equivalence and TLB equivalence on valid states. Formally, we define two valid states s_1 and s_2 to be cache equivalent for *osi*, written $s_1 \equiv_{osi}^{cache} s_2$, iff *osi* is the active OS in both states and the caches hold equal values for all accessible virtual addresses *va* that are in the domain of the cache of both states, i.e. for all virtual address *va* and pages p_1 and p_2

$$s_1.active\_os = s_2.active\_os = osi \;\rightarrow\; os\_accessible(va) \;\rightarrow\; s_1.cache[va] = p_1 \;\rightarrow\; s_2.cache[va] = p_2 \;\rightarrow\; p_1 = p_2\,.$$

Note that we do not require that the domains of both caches coincide, as it would invalidate the following lemma.

▶ **Lemma 6** (Cache equivalence)**.**
$\forall\ (s_1\ s_2 : State)\ (osi : os\_ident),$
$valid\_state(s_1) \;\rightarrow\; valid\_state(s_2) \;\rightarrow\; s_1 \equiv_{osi} s_2 \;\rightarrow\; s_1 \equiv_{osi}^{cache} s_2\,.$

The notion of TLB equivalence is defined in a similar way. We say that two valid states s_1 and s_2 are TLB equivalent for *osi*, written $s_1 \equiv_{osi}^{tlb} s_2$, iff *osi* is the active OS in both states and for all accessible virtual addresses *va* that are in the domain of the TLB of both states, if the machine address $s_1.tlb[va]$ holds a page with RW memory content, then if *va* appears in $s_2.tlb$, it holds the same page, i.e. for all machine addresses ma_1 and ma_2, and page *pg*:

$$s_1.active\_os = s_2.active\_os = osi \;\rightarrow\; s_1.tlb[va] = ma_1 \;\rightarrow\; s_1.memory[ma_1] = pg \;\rightarrow\; \exists\ (val : Value), pg.page\_content = RW\ (Some\ val) \;\rightarrow\; s_2.tlb[va] = ma_2 \;\rightarrow\; s_2.memory[ma_2] = pg$$

and conversely. We have:

▶ **Lemma 7** (Tlb equivalence)**.**
$\forall\ (s_1\ s_2 : State)\ (osi : os\_ident),$
$valid\_state(s_1) \;\rightarrow\; valid\_state(s_2) \;\rightarrow\; s_1 \equiv_{osi} s_2 \;\rightarrow\; s_1 \equiv_{osi}^{tlb} s_2\,.$

We write $s_1 \equiv_{osi}^{cache,tlb} s_2$ as a shorthand for $s_1 \equiv_{osi} s_2 \;\wedge\; s_1 \equiv_{osi}^{cache} s_2 \;\wedge\; s_1 \equiv_{osi}^{tlb} s_2$. We can now generalize the unwinding lemmas of [4]: the first lemma states that equivalence is preserved by the execution of all actions that do not generate errors.

▶ **Lemma 8** (Step-consistent unwinding lemma)**.**
$\forall\ (s_1\ s_1'\ s_2\ s_2' : State)\ (a : Action)\ (osi : os\_ident),$
$s_1 \equiv_{osi} s_2 \;\rightarrow\; os\_action(s_1, a, osi) \;\rightarrow\; os\_action(s_2, a, osi) \;\rightarrow\; s_1 \xrightarrow{a/ok} s_1' \;\rightarrow\; s_2 \xrightarrow{a/ok} s_2' \;\rightarrow\; s_1' \equiv_{osi}^{cache,tlb} s_2'\,.$

where $os\_action(s, a, osi)$ denote that action a is an action successfully executed by the OS *osi* in the state s; in particular, its execution does not cause an error. Note that an execution that fails does not generate a change in the system state.

The second lemma states that execution does not alter the state of non-active OSs, or active OS if it performs an execution that fails.

▶ **Lemma 9** (Locally preserves unwinding lemma)**.**
$\forall\ (s\ s' : State)\ (a : Action)\ (r : Response)\ (osi : os\_ident),$
$\neg\ os\_action(s, a, osi) \;\rightarrow\; s \xrightarrow{a/r} s' \;\rightarrow\; s \equiv_{osi}^{cache,tlb} s'\,.$

4.2 OS isolation in execution traces

The extension to traces of the relation one-step execution with error management is defined as follows: an execution trace is defined as a stream (an infinite list) of states that are related by the transition relation $\xrightarrow{a/r}$, i.e. an object of the form

$$s_0 \xrightarrow{a_0/r_0} s_1 \xrightarrow{a_1/r_1} s_2 \xrightarrow{a_2/r_2} s_3 \ldots$$

In the sequel, we let $t[i]$ denote the i-th state of a trace t and we use $s \xrightarrow{a/r} t$ to denote the trace obtained by prepending the valid execution step $s \xrightarrow{a/r} t[0]$ to a trace t. We let $Trace$ define the type of these traces. Isolation properties are eventually expressed on execution traces, rather than execution steps.

Non-influencing execution (errors)

Using the unwinding lemmas previously presented, one can establish a non-influence result in the style of [25]. We define for each operating system *osi* a predicate *same_os_actions* stating that two traces have the same set of actions w.r.t. *osi*; so that two traces are related iff they perform the same valid *osi*-actions. Then we define two traces t_1 and t_2 to be *osi-equivalent*, written $t_1 \approx_{osi,cache,tlb} t_2$, co-inductively by the following rules:

$$\frac{t_1 \approx_{osi,cache,tlb} t_2 \quad \neg\ os\_action(s, a, osi)}{(s \xrightarrow{a/r} t_1) \approx_{osi,cache,tlb} t_2}$$

$$\frac{t_1 \approx_{osi,cache,tlb} t_2 \quad \neg\ os\_action(s, a, osi)}{t_1 \approx_{osi,cache,tlb} (s \xrightarrow{a/r} t_2)}$$

$$\frac{t_1 \approx_{osi,cache,tlb} t_2 \quad os\_action(s_1, a, osi) \quad os\_action(s_2, a, osi) \quad s_1 \equiv_{osi}^{cache,tlb} s_2}{(s_1 \xrightarrow{a/ok} t_1) \approx_{osi,cache,tlb} (s_2 \xrightarrow{a/ok} t_2)}$$

▶ **Theorem 10** (OS isolation)**.**
$\forall\ (t_1\ t_2 : Trace)\ (osi : os\_ident)$,
$same\_os\_actions(osi, t_1, t_2) \rightarrow (t_1[0] \equiv_{osi} t_2[0]) \rightarrow t_1 \approx_{osi,cache,tlb} t_2$.

OS isolation formally establishes that two traces are *osi*-equivalent if they have the same set of *osi*-actions and if their initial states are *osi*-equivalent. The proof of OS isolation is based on co-induction principles and on the previous unwinding lemmas. Note that the definition of *osi*-equivalent traces conveniently generalizes the notion used in [4] (by allowing related traces to differ in the number of actions executed by other OSs) and extends that presented in [5] considering executions with error management. In particular, Theorem 10 states that the OS isolation property introduced in [5] is also valid in the context of executions that include error handling, considering that an *osi*-action is an action successfully executed by the OS *osi*.

Though it is left as future work, it is interesting to comment on the validity of isolation properties under other policies. On the one hand, the replacement policy for the cache and the TLB is left abstract in our model, so any reasonable algorithm will preserve these properties (as embodied e.g. in the definition of *cache_add* in Section 3.1). On the other hand, we have fixed a *write-through* policy for the main memory: this policy entails that updates to memory pages are done simultaneously to the cache and main memory, and we have used throughout the development the invariant property that cache data is included in the memory. This inclusion property will not hold if we were to use a *write-back* policy,

in which written entries are marked dirty and updates to main memory are done when a page is removed from the cache. We believe that it remains possible to prove strong isolation properties under the *write-back* policy, since page values, even if different in memory, will be equal if we consider the cache and memory together.

Finally, the flushing policy is assumed to be a total flush on switch and local switch execution. An alternative would be to tag cache (and TLB) entries with the virtual spaces allowed to access the entry. This will not have as much impact on the current model as the write policy, though changes will need to be done to the cache definition to include the tags. Isolation properties would still hold, given correct semantics of access control of cache entries.

5 Related work and conclusion

Thanks to recent advances in verification technology, it is now becoming feasible to verify formally realistic specifications and implementations of operating systems. A recent account of existing efforts can be found in the surveys [18, 30]. Many of these works focus on functional correctness of the hypervisor; one notable exception is [24], which proves that the seL4 microkernel guarantees information flow security; this work builds on a proof of integrity [29] and a proof of correctness and culminates a 30+ man-year verification effort. In addition, many of these works do not consider cache, which is a distinctive focus of our work. On the other hand, most of these works focus on implementations, and provide an explicit treatment of errors – that was missing in our earlier work [5].

Moving away from OS verification, many works have addressed the problem of relating inductively defined relations and executable functions, in particular in the context of programming language semantics. For instance, Tollitte *et al* [33] show how to extract a functional implementation from an inductive specification in the Coq proof assistant. Similar approaches exist for Isabelle, see e.g. [8]. Earlier, alternative approaches such as [2, 6] aim to provide reasoning principles for executable specifications.

We have enhanced the idealized model of virtualization considered in [5] with an explicit treatment of errors, and showed that OS isolation is preserved in this setting. Moreover we have implemented an executable specification that realizes the axiomatic semantics used in [5]. The formal development in this paper is about 15 kLOC of Coq, where 8k correspond to the verified executable specification and 7k to the OS isolation proof on the extended model with errors. In [28] we derive two certified hypervisor implementations, using the extraction mechanism of Coq [22, 21], in functional languages Haskell and OCaml.

In future work, we intend to implement alternative executable semantics for different models of cache and policies. Moreover, we plan to use our extended model as a basis for investigating whether error management can lead to side-channels.

Acknowledgements. The authors want to thank TYPES reviewers for helpful feedback on the paper.

The work of Gilles Barthe has been partially funded by European Project FP7 256980 NESSoS, Spanish project TIN2009-14599 DESAFIOS 10 and Madrid Regional project S2009TIC-1465 PROMETIDOS and the work of Gustavo Betarte, Juan Diego Campo and Carlos Luna by Uruguayan project CSIC-Convocatoria 2012, Proyectos I + D, VirtualCert – Fase II.

References

1 June Andronick. *Modélisation et Vérification Formelles de Systèmes Embarqués dans les Cartes à Microprocesseur – Plate-Forme Java Card et Système d'Exploitation*. PhD thesis, Université Paris-Sud, 2006.

2 Antonia Balaa and Yves Bertot. Fix-point equations for well-founded recursion in type theory. In Mark Aagaard and John Harrison, editors, *TPHOLs*, volume 1869 of *LNCS*, pages 1–16. Springer, 2000.

3 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 19th ACM Symposium on Operating Systems Principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

4 G. Barthe, G. Betarte, J. D. Campo, and C. Luna. Formally verifying isolation and availability in an idealized model of virtualization. In *FM 2011*, pages 231–245. Springer-Verlag, 2011.

5 G. Barthe, G. Betarte, J. D. Campo, and C. Luna. Cache-Leakage Resilient OS Isolation in an Idealized Model of Virtualization. In *CSF 2012*, pages 186–197, 2012.

6 G. Barthe, J. Forest, D. Pichardie, and V. Rusu. Defining and reasoning about recursive functions: A practical tool for the coq proof assistant. In M. Hagiya and P. Wadler, editors, *FLOPS*, volume 3945 of *LNCS*, pages 114–129. Springer, 2006.

7 Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. *SIGPLAN Not.*, 44(1):90–101, January 2009.

8 Stefan Berghofer, Lukas Bulwahn, and Florian Haftmann. Turning inductive into equational specifications. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *TPHOLs*, volume 5674 of *LNCS*, pages 131–146. Springer, 2009.

9 Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.

10 G. Betarte, E. Giménez, C. Loiseaux, and B. Chetali. FORMAVIE: Formal Modeling and Verification of the Java Card 2.1.1 Security Architecture. In *Proceedings of eSmart'02*, 2002.

11 Boutheina Chetali and Quang-Huy Nguyen. About the world-first smart card certificate with eal7 formal assurances. Slides 9th ICCC, Jeju, Korea, September 2008.

12 M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

13 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.

14 Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

15 Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *Conference on Computer Logic*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 1988.

16 J.-Y. Hwang, S.-B. Suh, S.-K. Heo, C.-J. Park, J.-M. Ryu, S.-Y. Park, and C.-R. Kim. Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones. In *5th IEEE Consumer and Communications Networking Conference*, 2008.

17 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.

18 Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas

Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In *SOSP 2009*, pages 207–220. ACM, 2009.

19 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.

20 Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52:107–115, July 2009.

21 P. Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq*. PhD thesis, Université Paris-Sud, July 2004.

22 Pierre Letouzey. A New Extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002*, volume 2646 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

23 Silvio Micali and Leonid Reyzin. Physically observable cryptography (extended abstract). In *TCC 2004*, pages 278–296, 2004.

24 T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. seL4: from General Purpose to a Proof of Information Flow Enforcement. In *Proc. of the 2013 IEEE Symp. on Security and Privacy (SP'13)*, pages 415–429, 2013.

25 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.

26 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.

27 C. Paulin-Mohring. Inductive definitions in the system coq - rules and properties. In M. Bezem and J. F. Groote, editors, *1st Int. Conf. on Typed Lambda Calculi and Applications*, volume 664 of *LNCS*, pages 328–345. Springer-Verlag, 1993.

28 The VirtualCert project. Supporting Coq formalization. See `http://www.fing.edu.uy/inco/grupos/gsi/proyectos/virtualcert.php`.

29 Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. seL4 enforces integrity. In *ITP 2011*, Nijmegen, The Netherlands, 2011.

30 Zhong Shao. Certified software. *Commun. ACM*, 53(12):56–66, 2010.

31 The Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2012.

32 T. Terauchi and A. Aiken. Secure information flow as a safety problem. In C. Hankin and I. Siveroni, editors, *Proceedings of SAS'05*, volume 3672 of *LNCS*, pages 352–367. Springer-Verlag, 2005.

33 Pierre-Nicolas Tollitte, David Delahaye, and Catherine Dubois. Producing certified functional code from inductive specifications. In Chris Hawblitzel and Dale Miller, editors, *CPP*, volume 7679 of *LNCS*, pages 76–91. Springer, 2012.

34 Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *J. Cryptology*, 23(1):37–71, 2010.

A Appendix

A.1 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 the following properties:

- if the active OS is in *running* mode then no hypercall requested by it is pending;
- if the hypervisor or a trusted OS (respectively untrusted OS) is running the processor must be in supervisor (respectively user) mode;
- the hypervisor maps an OS physical address to a machine address owned by that same OS. This mapping is also injective;
- all page tables of an OS o map virtual addresses to pages owned by o;
- the current page table of any OS is owned by that OS;
- any machine address 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 cache keys are related in a page table mapping of the memory;
- all cache pages have the same owner and type as those in machine memory;
- if *va* is translated into *ma* according to the TLB, then the machine address *ma* is associated to *va* in the active memory mapping.

All properties have a straightforward interpretation in our model. For example, the first property is captured by the proposition:

$$\forall\ osi : os\_ident,\ trusted\_os(osi) \rightarrow (s.oss[osi]).hcall\ =\ None\,.$$

A.2 Actions

Table 3 summarises the complete set of actions specified in the model, and their effects.

A.3 Observational equivalence of states

We say that two states s_1 and s_2 are *osi-equivalent*, written $s_1 \equiv_{osi} s_2$, iff:

- *osi* is the active OS in both states and the processor mode is the same, or the active OS is different to *osi* in both states;
- *osi* has the same hypercall in both states, or no hypercall in both states;
- the current page tables of *osi* are the same in both states;
- all page table mappings of *osi* that map a virtual address to a RW page in one state, must map that address to a page with the same content in the other;
- 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 content at these machine addresses be the same in both states, if it corresponds to an RW page.

Table 3 Full set of actions.

`read` *va*	A guest OS reads virtual address *va*.
`read_hyper` *va*	The hypervisor reads virtual address *va*.
`write` *va val*	A guest OS writes value *val* in virtual address *va*.
`write_hyper` *va val*	The hypervisor writes value *val* in virtual address *va*.
`new_tr` *va pa*	The virtual address *va* is mapped to the machine address *ma* in the memory mapping of the trusted active OS, where *pa* translates to *ma* for the active OS.
`new_untr` *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*.
`new_hyper` *va ma*	The hypervisor adds a new ordered pair to the current memory mapping of the active OS (mapping virtual address *va* to the machine address *ma*) for his own purposes.
`del_tr` *va*	The trusted active OS deletes the ordered pair that maps virtual address *va* from its memory mapping.
`del_untr` *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*.
`del_hyper` *va*	The hypervisor deletes (for its own purposes) the ordered pair that maps virtual address *va* from the current memory mapping of the active OS.
`switch` *o*	The hypervisor sets *o* to be the active OS.
`lswitch_tr` *pa*	The trusted active OS changes its current memory mapping to be the one located at physical address *pa*. This action corresponds to a traditional context switch by the active OS.
`lswitch_untr` *o pa*	The hypervisor changes the current memory mapping of the untrusted active OS, to be the one located at physical address *pa*.
`silent`	Represents the silent action (the system does not advertise any effects).
`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.
`pin_tr` *pa t*	The memory page that corresponds to physical address *pa* (for the active OS) is registered and classified with type *t*.
`pin_untr` *o pa t*	The memory page that corresponds to physical address *pa* (for untrusted OS *o*) is registered and classified with type *t*.
`unpin_tr` *pa*	The memory page that corresponds to physical address *pa* (for the active OS) is un-registered.
`unpin_untr` *o pa*	The memory page that corresponds to physical address *pa* (for the untrusted OS *o*) is un-registered.