Formal analysis of security models for mobile devices, virtualization platforms, and other critical systems

Grupo de Seguridad Informática

InCo, Facultad de Ingeniería, Universidad de la República, Uruguay

2017

The Calculus of (Co)Inductive Constructions (CIC) and Coq

CIC is an extension of the simple-typed lambda calculus with:

- Polymorphic types $[(\lambda\ x\ .\ x) : A \to A]$
- Higher-order types $[A \to A : * : \Box]$
- Dependent types $[(\lambda\ a : A\ .\ f\ a) : (\forall\ a : A\ .\ B_a)]$

- Implemented in Coq
  Type checker + Proof assistant
- Can encode higher-order predicate logic
- (Co)Inductive definitions
- Curry-Howard isomorphism

$$\begin{array}{lcl} \text{types} & \leftrightarrow & \text{propositions} \\ \text{terms} & \leftrightarrow & \text{proofs} \end{array}$$

Part I

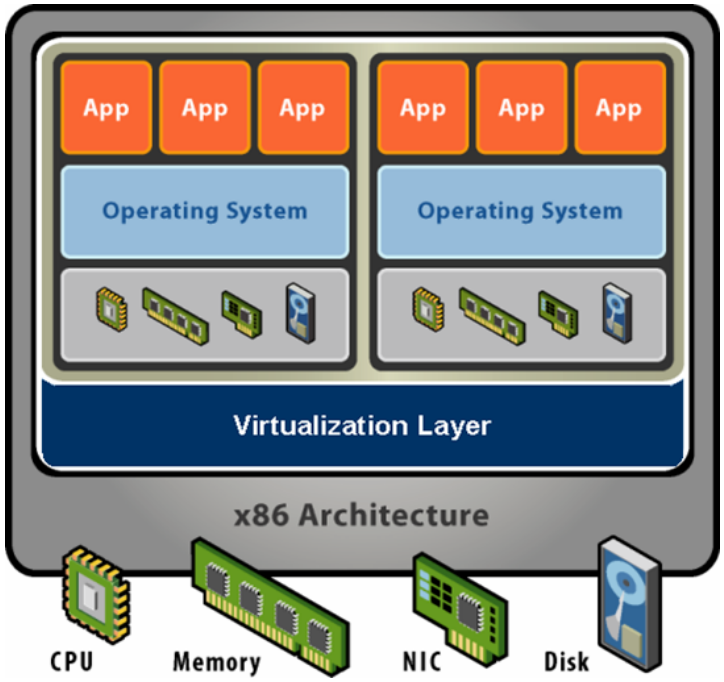
VirtualCert

OS verification

- OS verification since 1970
  - Hand written proofs
  - Type systems and program logics
  - Proof assistants
- OS verification is the next frontier
  - Tremendous advances in proof assistant technology
  - PL verification is becoming ubiquitous
- Flagship projects:
  - L4.verified: formal verification of seL4 kernel (G. Klein et al, NICTA)
  - Hyper-V: formal verification of Microsoft hypervisor (E. Cohen et al, MSR)

Virtualization

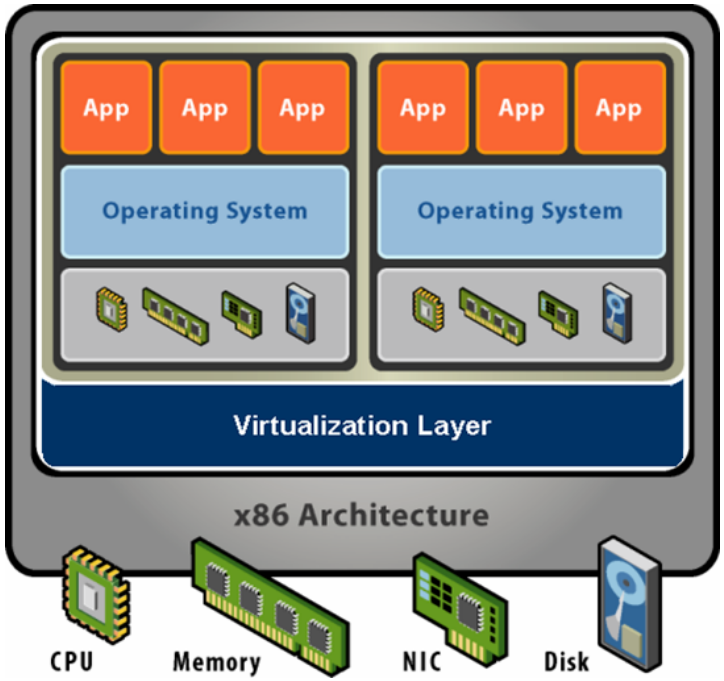
bare-metal hypervisors



- Allow several operating systems to coexist on commodity hardware
- Provide support for multiple applications to run seamlessly on the guest operating systems they manage
- Provide a means to guarantee that applications with different security policies can execute securely in parallel

Virtualization

bare-metal hypervisors



- Allow several operating systems to coexist on commodity hardware
- Provide support for multiple applications to run seamlessly on the guest operating systems they manage
- Provide a means to guarantee that applications with different security policies can execute securely in parallel

- They are increasingly used as a means to improve system flexibility and security
  - protection in safety-critical and embedded systems
  - secure provisioning of infrastructures in cloud computing

Hypervisors are a priority target of formal specification and verification

Motivation and challenge

- Main focus of L4.verified and Hyper-V on functional correctness
- We focus on non-functional properties:
  - Isolation
  - Transparency
  - Availability (maximizing resources under constraints)

  Both properties go beyond safety:
  - Isolation and transparency are 2-safety properties
  - Availability is a liveness property

Motivation and challenge

- Main focus of L4.verified and Hyper-V on functional correctness
- We focus on non-functional properties:
  - Isolation
  - Transparency
  - Availability (maximizing resources under constraints)

  Both properties go beyond safety:
  - Isolation and transparency are 2-safety properties
  - Availability is a liveness property
- We reason about classes of systems

Idealized models vs. implementations

Reasoning about implementations

- Give the strongest guarantees
- Is feasible for *some* exokernels and hypervisors
- May be feasible for *some* baseline properties of *some* systems
- Is out of reach in general (Linux Kernel)
- May not be required for evaluation purposes

Idealized models vs. implementations

Reasoning about implementations

- Give the strongest guarantees
- Is feasible for *some* exokernels and hypervisors
- May be feasible for *some* baseline properties of *some* systems
- Is out of reach in general (Linux Kernel)
- May not be required for evaluation purposes

Idealized models provide the right level of abstraction

- Many details of behavior are irrelevant for specific property
- Idealization helps comparing different alternatives
- Proofs are more focused, and achievable within reasonable time

Our focus: Xen on ARM

A popular bare-metal hypervisor initially developed at U. Cambridge

Architecture

A computer running the Xen hypervisor contains three components:

- The Xen Hypervisor (software component)
- The privileged Domain (*Dom*0): privileged guest running on the hypervisor with direct hardware access and management responsibilities
- Multiple Unprivileged Domain Guests (*DomU*): unprivileged guests running on the hypervisor, and executing hypercalls (access to services mediated by the hypervisor)

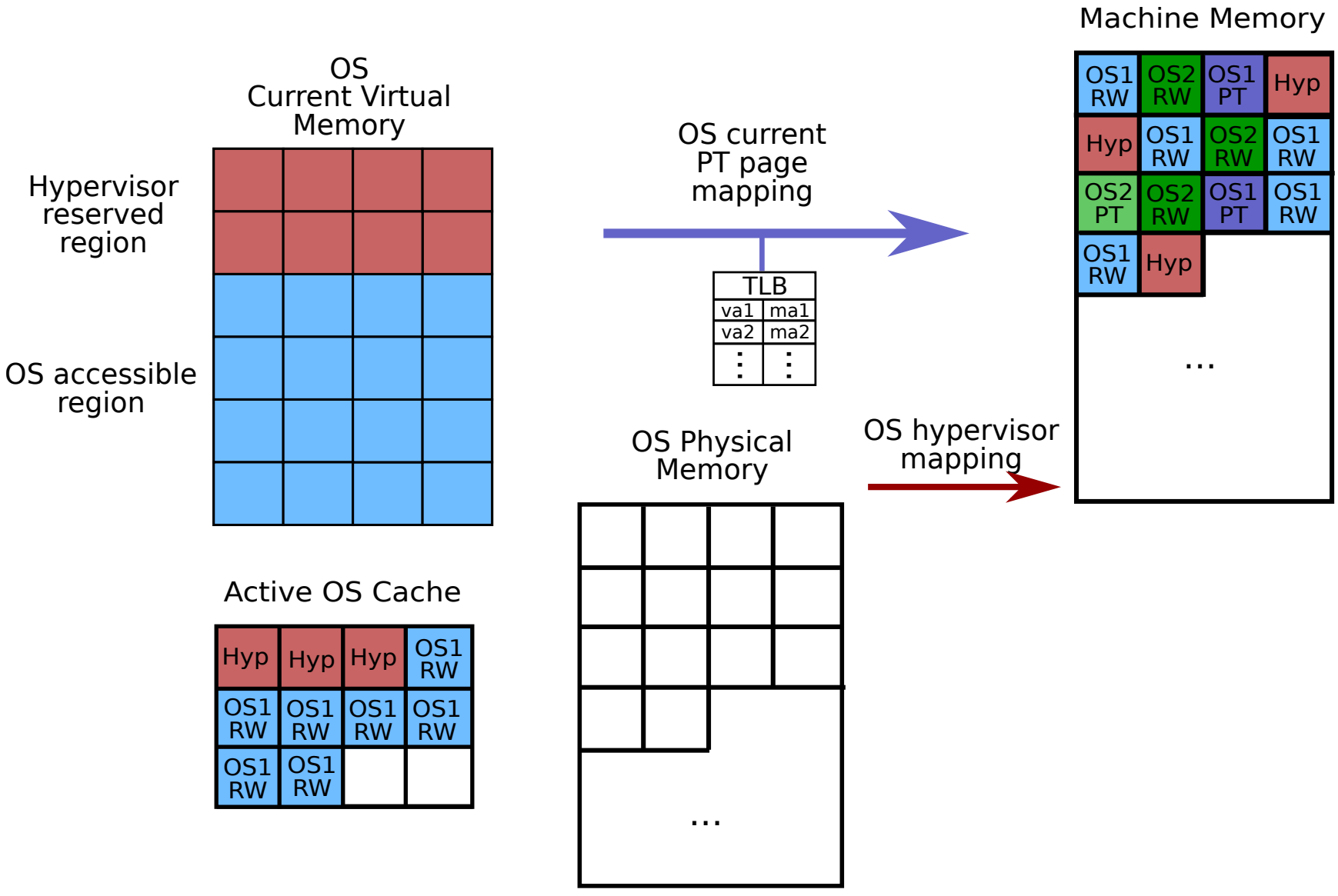
Xen on ARM

- Suggested during initial collaboration with VirtualLogix (now Red Bend Software)
- In turn, determines some modelling choices, e.g. for the cache

VirtualCert - Idealized model

- Abstract model written in Coq
- Focus on memory management
- Model of the hypervisor: based on Xen
- Model of the host machine: based on ARM

Memory model



States

$$
\begin{array}{lll}
State \stackrel{\text{def}}{=} \{ & active\_os & : os\_ident, \\
& aos\_exec\_mode & : exec\_mode, \\
& aos\_activity & : os\_activity, \\
& oss & : os\_ident \mapsto os\_info, \\
& hypervisor & : os\_ident \mapsto (padd \mapsto madd), \\
& memory & : madd \mapsto page \\
& cache & : vadd \mapsto_{size\_cache} page, \\
& tlb & : vadd \mapsto_{size\_tlb} madd \}
\end{array}
$$

OS information and pages

$$os\_info \stackrel{\text{def}}{=} \{ curr\_page : padd, hcall : option\ Hyper\_call \}$$

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

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

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

Execution: State transformers

read *va*	Guest OS reads virtual address *va*.
write *va val*	Guest OS writes value *val* in *va*.
read_hyper *va*	Hypervisor reads virtual address *va*.
write_hyper *va val*	Hypervisor writes value *val* in virtual address *va*.
hcall *c*	Guest OS requires privileged service *c* to the hypervisor.
new *o va pa*	Hypervisor extends *os* memory with $va \mapsto ma$.
del *o va*	Hypervisor deletes mapping for *va* from current memory mapping of *o*.
lswitch *o pa*	Hypervisor changes the current memory mapping of the active OS, to be the one located at physical address *pa*.
switch *o*	Hypervisor sets *o* to be the active OS.
ret_ctrl	Returns control to the hypervisor.
chmod	Hypervisor changes execution mode from supervisor to user mode, and gives control to the active OS.
page_pin *o pa t*	Registers memory page of type *t* at address *pa*.
page_unpin *o pa*	Memory page at *pa* is un-registered.

Semantics

Axiomatic specification

- Pre-condition $Pre : State \rightarrow Action \rightarrow Prop$
- Post-condition $Post : State \rightarrow Action \rightarrow State \rightarrow Prop$
- Focus on normal execution: no semantics for error cases
- Alternatives (write through/write back, replacement and flushing policies)
- One step execution:

$$s \xrightarrow{a} s' \stackrel{\text{def}}{=} valid\_state(s) \wedge Pre\ s\ a \wedge Post\ s\ a\ s'$$

- Traces:

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \ldots$$

- Valid state:
  - invariant under execution
  - key to isolation results

Valid state

Many conditions, e.g:

- if the hypervisor or a trusted OS is running the processor must be in supervisor mode
- if an untrusted OS is running the processor must be in user mode
- 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
- the current page table of any OS is owned by that OS
- 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
- ...

Semantics

Write Action

$$
\begin{aligned}
&\mathit{Pre}\ s\ (\mathit{write}\ \mathit{va}\ \mathit{val}) \stackrel{\text{def}}{=} \exists \mathit{ma}, \mathit{pg} \\
&\quad \mathit{os\_accessible}(\mathit{va}) \wedge \\
&\quad s.\mathit{aos\_activity} = \mathit{running} \wedge \\
&\quad \mathit{va\_mapped\_to\_ma}(s, \mathit{va}, \mathit{ma}) \wedge \\
&\quad \mathit{va\_mapped\_to\_pg}(s, \mathit{va}, \mathit{pg}) \wedge \\
&\quad \mathit{is\_RW}(\mathit{pg})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Post}\ s\ (\mathit{write}\ \mathit{va}\ \mathit{val})\ s' \stackrel{\text{def}}{=} \\
&\quad \text{let } (\mathit{new\_pg} : \mathit{page} = \langle \mathit{RW}(\mathit{Some}\ \mathit{val}), \mathit{pg.page\_owned\_by} \rangle) \text{ in} \\
&\quad s' = s \cdot \left[\begin{array}{rcl} \mathit{mem} & := & (s.\mathit{memory}[\mathit{ma} := \mathit{new\_pg}]), \\ \mathit{cache} & := & \mathit{cache\_add}(\mathit{fix\_cache\_syn}(s.\mathit{cache}, \mathit{ma}), \mathit{va}, \mathit{new\_pg}), \\ \mathit{tlb} & := & \mathit{tlb\_add}(s.\mathit{tlb}, \mathit{va}, \mathit{ma}) \end{array} \right]
\end{aligned}
$$

Equivalence w.r.t. an OS

Two states s_1 and s_2 are *osi*-equivalent, written $s_1 \equiv_{osi} s_2$, iff:

1. *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
2. *osi* has the same hypercall in both states, or no hypercall in both states
3. the current page tables of *osi* are the same in both states
4. 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
5. 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

Isolation properties

Read isolation

No OS can read memory that does not belong to it

Isolation properties

Read isolation

No OS can read memory that does not belong to it

Write isolation

An OS cannot modify memory that it does not own

Isolation properties

Read isolation

No OS can read memory that does not belong to it

Write isolation

An OS cannot modify memory that it does not own

OS isolation (on traces)

$$\forall\ (t_1\ t_2 : \mathit{Trace})\ (\mathit{osi} : \mathit{os\_ident}),$$
$$(t_1[0] \equiv_{\mathit{osi}}\ t_2[0]) \rightarrow$$
$$\mathit{same\_os\_actions}(\mathit{osi}, t_1, t_2) \rightarrow$$
$$\Box(\equiv_{\mathit{osi}}, t_1, t_2)$$

Availability

- IF the hypervisor only performs `chmod` actions whenever no hypercall is pending
- AND the hypervisor returns control to guest operating systems infinitely often
- THEN no OS blocks indefinitely waiting for its hypercalls to be attended

$$\forall\,(t : \mathit{Trace}), \neg\, \mathit{hcall}(t[0]) \rightarrow$$
$$\Box(\mathit{chmod\_nohcall}, t) \rightarrow$$
$$\Box(\Diamond\, \neg\, \mathit{hyper\_running}, t) \rightarrow$$
$$\Box(\Diamond\, \neg\, \mathit{hcall}, t)$$

Fairness and other properties

- Does not guarantee that every OS will eventually get attended
- Many other policies may be considered

Part II

A certified idealized hypervisor

Implementation in Coq

- We present an implementation of an hypervisor in the programming language of Coq
- 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

Error management

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

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*

Table: Preconditions and error codes

Executions with error management

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

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

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

Executions with error management

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

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

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

Lemma (Validity is invariant)

$$\forall\ (s\ s' : State)(a : Action)(r : Response),\ valid\_state(s) \to s \xrightarrow{a/r} s' \to valid\_state(s')$$

Action execution

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

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

Execution of `write` action

```
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.

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.
```

Effect of `write` execution

```
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
```

Effect of `write` execution (2)

```
  | 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.
```

Soundness

Theorem (Soundness of hypervisor implementation)

$$\forall\ (s : State)\ (a : Action),\ valid\_state(s) \rightarrow$$
$$s \xrightarrow{a/step(s,a).resp} step(s, a).st$$

Soundness

Theorem (Soundness of hypervisor implementation)

$$\forall\ (s : State)\ (a : Action),\ valid\_state(s) \rightarrow$$
$$s \xrightarrow{a/step(s,a).resp} step(s, a).st$$

Lemma (Soundness of error execution)

$$\forall\ (s : State)\ (a : Action),$$
$$valid\_state(s) \rightarrow\ \neg Pre(s, a) \rightarrow\ \exists\ (ec : ErrorCode),$$
$$step(s, a).st = s\ \wedge\ step(s, a).resp = ec\ \wedge\ ErrorMsg(s, a, ec)$$

Lemma (Soundness of valid execution)

$$\forall\ (s : State)\ (a : Action),\ valid\_state(s)\ \rightarrow\ Pre(s, a)\ \rightarrow$$
$$s \xrightarrow{a/ok} step(s, a).st\ \wedge\ step(s, a).resp = ok$$

Non-influencing execution (errors)

Traces

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

Non-influencing execution (errors)

Traces

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

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

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

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

Non-influencing execution (errors)

Traces

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

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

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

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

Cache and TLB equivalences

$$s_1 \equiv_{osi}^{cache,tlb} s_2 \quad \text{iff} \quad s_1 \equiv_{osi} s_2 \wedge s_1 \equiv_{osi}^{cache} s_2 \wedge s_1 \equiv_{osi}^{tlb} s_2$$

OS isolation in execution traces (with errors)

Theorem (OS isolation)

$$\forall\ (t_1\ t_2 : \mathit{Trace})\ (\mathit{osi} : \mathit{os\_ident}),$$
$$\mathit{same\_os\_actions}(\mathit{osi}, t_1, t_2) \rightarrow$$
$$(t_1[0] \equiv_{\mathit{osi}} t_2[0]) \rightarrow t_1 \approx_{\mathit{osi},\mathit{cache},\mathit{tlb}} t_2$$

OS isolation in execution traces (with errors)

Theorem (OS isolation)

$$\forall\ (t_1\ t_2 : \mathit{Trace})\ (\mathit{osi} : \mathit{os\_ident}),$$
$$\mathit{same\_os\_actions}(\mathit{osi}, t_1, t_2) \rightarrow$$
$$(t_1[0] \equiv_{\mathit{osi}} t_2[0]) \rightarrow t_1 \approx_{\mathit{osi},\mathit{cache},\mathit{tlb}} t_2$$

Lemma (Locally preserves unwinding lemma)

$$\forall\ (s\ s' : \mathit{State})\ (a : \mathit{Action})\ (r : \mathit{Response})\ (\mathit{osi} : \mathit{os\_ident}),$$
$$\neg\ \mathit{os\_action}(s, a, \mathit{osi}) \ \rightarrow\ s \xrightarrow{a/r} s' \ \rightarrow\ s \equiv_{\mathit{osi}}^{\mathit{cache},\mathit{tlb}} s'$$

OS isolation in execution traces (with errors)

Theorem (OS isolation)

$$\forall\ (t_1\ t_2 : \mathit{Trace})\ (\mathit{osi} : \mathit{os\_ident}),$$
$$\mathit{same\_os\_actions}(\mathit{osi}, t_1, t_2) \rightarrow$$
$$(t_1[0] \equiv_{\mathit{osi}} t_2[0]) \rightarrow t_1 \approx_{\mathit{osi},\mathit{cache},\mathit{tlb}} t_2$$

Lemma (Locally preserves unwinding lemma)

$$\forall\ (s\ s' : \mathit{State})\ (a : \mathit{Action})\ (r : \mathit{Response})\ (\mathit{osi} : \mathit{os\_ident}),$$
$$\neg\ \mathit{os\_action}(s, a, \mathit{osi}) \rightarrow s \xrightarrow{a/r} s' \rightarrow s \equiv_{\mathit{osi}}^{\mathit{cache},\mathit{tlb}} s'$$

Lemma (Step-consistent unwinding lemma)

$$\forall\ (s_1\ s_1'\ s_2\ s_2' : \mathit{State})\ (a : \mathit{Action})\ (\mathit{osi} : \mathit{os\_ident}),$$
$$s_1 \equiv_{\mathit{osi}} s_2 \rightarrow \mathit{os\_action}(s_1, a, \mathit{osi}) \rightarrow \mathit{os\_action}(s_2, a, \mathit{osi}) \rightarrow$$
$$s_1 \xrightarrow{a/ok} s_1' \rightarrow s_2 \xrightarrow{a/ok} s_2' \rightarrow s_1' \equiv_{\mathit{osi}}^{\mathit{cache},\mathit{tlb}} s_2'$$

Work in progress

- Extension of the virtualization model to use a VIPT/PIPT cache and abstract replacement and write policies
- Using the model for reasoning about cache-based attacks and countermeasures
- **Multi-core !!!**