An idealized model of virtualization with stealth memory: complete action semantics and non-interference

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

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

Abstract. We present the complete formal semantics of an idealized model of virtualization which incorporates management of stealth memory (StealthCert). The paper provides a detailed account of the basic components of the idealized model focusing on the memory model and the notion of state that has been formalized as well as a formal axiomatic and executable semantics of an idealized hypervisor. An isolation theorem for StealthCert is discussed and a sketch of its proof, which has been developed using Coq, presented.

1 States

In this section we give an overview of the state of the idealized virtualization model, which includes the main memory of the platform, various kinds of memory spaces, and the cache and TLB. The most important component of the state is the memory model, which we proceed to describe.

In Figure 1 we show a high level diagram of the memory model, which involves three types of addressing modes and two address mappings.

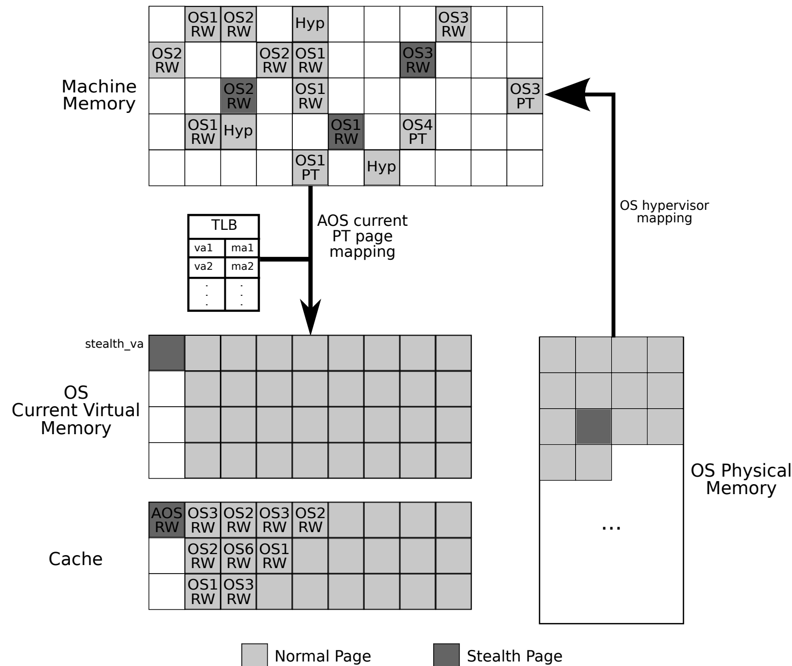

Fig. 1. Memory model of the platform

The *machine memory* is the real machine memory. A mechanism of page classification was introduced in order to cover concepts from virtualization platforms, in particular Xen [1]. The model considers that each

Va, Pa, Ma		virtual, physical and machine address
OSId		OS identifier
HC	$::= new \mid del \mid lswitch \mid pin \mid unpin \mid none$	hyper calls
OSData	$::=$ Pa $\times$ HC	OS data
GuestOSs	$::=$ OSId $\rightarrow$ OSData	guest OSs
OSActivity	$:= running \mid waiting$	exec modes
ActiveOS	$::=$ OSId $\times$ OSActivity	active OS
PageContent	$:= RW(\mathsf{Value}) \mid PT(\mathsf{Va} \rightarrow \mathsf{Ma}) \mid none$	page content
PageOwner	$:= Hyp \mid OS(\mathsf{OSId}) \mid none$	page owner
Page	$:=$ PageContent $\times$ PageOwner $\times$ Bool	memory page
Memory	$::=$ Ma $\rightarrow$ Page	memory map
HyperMap	$::=$ OSId $\rightarrow$ Pa $\rightarrow$ Ma	hypervisor map
CacheData	$:=$ Va $\times$ Ma $\mapsto$ Page	cache data
CacheIndex	$:=$ Va $\rightarrow$ Index	cache index
CacheHistory	$:=$ Index $\rightarrow$ Hist	cache history
Cache	$::=$ CacheData $\times$ CacheIndex $\times$ CacheHistory	VIPT cache
TLB	$::=$ Va $\mapsto$ Ma	TLB
SLST	$::=$ GuestOSs $\times$ ActiveOS $\times$ HyperMap $\times$ Memory $\times$ Cache $\times$ TLB	Stealth state

Fig. 2. Formal definition of the state

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 machine addresses (written *madd*) model real hardware memory on the host machine and are never directly accessed by the guest operating systems. In usual architectures with paging the addresses are composed of two parts: a page identifier and an offset inside the page. We will make an abstraction on this behaviour, not distinguishing between accesses at different offsets inside the page. All machine addresses will therefore refer to a different page, and we will only record the access to the memory page, not the specific data inside it. This gives us a good level of abstraction to reason about memory accesses, while still capturing the behaviour of actual implementations.

The *physical memory* is the one addressed by the kernel of the guest OS. In the Xen platform this is the type of addresses that the hypervisor exposes to the domains (the guest OSs in our model). The physical addresses (whose type is written *padd*) are 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 it is transparent to the guest operating systems.

The virtual addresses are used by applications running on guest 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. Moreover, each OS has a designated portion of its virtual address space that is reserved for the hypervisor to attend hypercalls. We say that a virtual address va is *accessible* by the OS ($os_accessible(va)$) 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*.

In Figure 2 we give the formalization of the platform state, which consists of a collection of components that we now proceed to describe.

Operating systems We define a type *os_ident* of identifiers for guest operating systems. 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. A hypercall is a privileged functionality exported by the hypervisor to the guest OSs.

Formally the information is captured by a mapping *oss* that associates OS identifiers with objects of type *os_data*.

Active OS and execution modes One of the operating systems is active at any given time (*active_os*). For this OS, in addition to its identifier, we register whether it is currently running or waiting for a hypercall to be resolved. Active OS execution mode is formalized by the type*os_activity*.

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) guest OSs execute in user mode while 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. After requesting the hypervisor to execute some service, the active guest OS will turn to processor execution mode *waiting* until the service is completed and the execution control returned, switching then its execution mode to *running*.

Mappings The mapping that associates, for each OS, machine addresses to physical ones is, in our model, managed by the hypervisor. This mapping, which is formalized as the component *hypervisor* of the state, might be treated differently by each specific virtualization platform. There are platforms in which this mapping is public and the OS is allowed to use machine addresses. The physical-to-machine address mapping is modified by the actions `page_pin` and `page_unpin`, as shall be described in Section 2.

The *memory* is formalized as a mapping from machine addresses to pages. A memory page consists of a page content and a page metadata. The content is either a readable/writable value, an OS page table (a mapping from virtual to machine addresses), or nothing. 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 guest 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. We consider an abstract type of values equipped with an equality relation, and we assume given a distinguished value $\perp$ when the value is undefined. In particular we abstract away implementation details such as encoding, size, etc.

The metadata contains a reference to the page owner (the hypervisor, an OS, or none) and a flag indicating whether the page can be cached or not.

Cache The figure also shows the cache, which is accessed by a pair of virtual and machine addresses (modeling a VIPT cache) and holds a subset of the memory pages. The size of the cache is bounded by a positive fixed constant K_c.

We also model historic access information as an abstract component of the cache. This is used by the replacement policy to decide the entry to be evicted from the cache.

In VIPT caches indices of the cache are derived from the virtual addresses, but each entry is tagged with the machine address. This avoids the need of flushing the cache on every context switch, and therefore requires less software management.

The cache can be seen as a collection of data blocks or *cache lines* (which are pages in our model) that are accessed by cache indices. There is a mapping *va2index* from virtual addresses to cache indices. Since caches are usually set associative, there are many virtual addresses that map to the same index. All data that is accessed using the same index is called a *cache line set*. As defined in [2] we assume that the *inertia* property holds for the cache. This property states that after adding an entry to the cache in a virtual address *va* and replacing some entry, the evicted address is in the same cache line set as *va*.

We select one cache index and one particular virtual address (*stealth_va*) in its cache line set for stealth use. All other virtual addresses in that cache line set are reserved and cannot be used either by the guest

operating systems or the hypervisor. Note that the use of only one stealth page is usually not an important restriction, given the intended usage of the stealth memory mechanism: the amount of information that is usually stored in stealth pages is small. Furthermore, it is relatively straightforward to extend the definitions to use a set of stealth addresses.

TLB The TLB is used in conjunction with the current page table of the active OS to speed up translation of virtual to machine addresses. The TLB is flushed on context switch and updates are done simultaneously in the page table, so its management is simpler than the cache. Therefore we do not record the TLB access history, as it is not necessary to write back evicted TLB entries. The size of the TLB is bounded by a positive fixed constant K_t.

1.1 Access functions

We use some helper functions to manipulate the components of the state. There is, for example, a function *cache_add* that is used to add entries in the cache. It returns the new cache and an optional entry selected for replacement. The function *cache_add* is parameterized by an abstract replacement policy that determines which elements are evicted from a full cache, and guarantees that the *inertia* property, as defined in [2], holds for the cache: when adding an entry to the cache in a virtual address *va*, if an eviction occurs, the evicted address is in the same cache line set as *va*.

1.2 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:

1. if the active OS is in running mode then no hypercall requested by it is pending;
2. if the hypervisor is *running*, the processor must be in *supervisor* mode, and if an OS is *running*, the processor must be in *user* mode;
3. the hypervisor maps an OS physical address to a machine address owned by that same OS. This mapping is also injective;
4. 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;
5. the current page table of any OS is owned by that OS;
6. 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;
7. memory pages from different virtual addresses mapped to the same machine address (*synonym* problem) are marked as *non-cacheables*;
8. all cache keys are related in a page table mapping of the memory;
9. all cache pages have the same owner and type (readable/writable) than those in machine memory;
10. if *va* is translated into *ma* according to the TLB, then the machine address *ma* is associated to *va* in the active memory mapping;
11. the current stealth page of the active OS (which is a readable/writable page) is always cached;
12. if an entry is cached in the stealth page line (which is unique), it must be the stealth page of the active OS.

All properties have a straightforward interpretation in our model. Valid states are invariant under execution, as shall be shown later.

2 Action Semantics

The operational semantics of the platform is modelled as a labelled transition system:

$$s \overset{a}{\hookrightarrow} s'$$

where s, s' range over StealthCert states and a ranges over actions. Informally, such a transition indicates that the execution of the action a in an initial state s leads to a new state s'.

Actions can be classified as follows: i) access, from an OS or the hypervisor, to memory pages (`read`, `read_hyper`, `write` and `write_hyper`); ii) update of page tables by the hypervisor on demand of an OS (`new`, `new_sm` and `del`); iii) context switches, or change of the active OS or the current page table of an OS by the hypervisor (`switch` and `lswitch`); iv) hypervisor call (`hcall`); v) changes of the execution mode (`chmod` and `ret_ctrl`); and vi) changes in the hypervisor memory mapping, which are performed by the hypervisor on demand of an OS (`pin` and `unpin`); vii) a silent action (`silent`), that models behaviour that does not affect the state.

Actions performed by a guest operating system can be classified as stealth or non-stealth, depending on the type of virtual addresses they access. The set of stealth actions include memory accesses to the stealth page (reads and writes), and memory allocation (with the `new_sm` action) or de-allocation of pages that are pointed by *stealth_va*. All other actions are considered *non-stealth*.

We define the effect of actions to classify the observations that can be drawn from an action execution. Stealth action have no effect, and non stealth actions have as effect the same action, except for the write, which only has as the effect parameter the virtual address and not the value. Note that two actions have the same effect if and only if, the actions are either two arbitrary stealth actions, the same non strealth action, or two writes to the same address but with arbitrary values. In section 3.3 this notion of effects is used to express the isolation property.

Figure 3 presents the platform actions and indicates their effect.

In what follows the semantics of actions shall be presented using the following layout:

Action `name` *args*

Informal description of the action behavior

Rule

Formal description of the behavior using natural semantics.

Precondition

Informal description of the conditions that must be satisfied for a correct execution of the action.

Postcondition

Informal description of the effect of the action execution

Notes

Remarks concerning the action

2.1 Memory accesses

Action `read` *va*

Guest OS reads the data in *va*

Rule

Action	Informal description	Effect
`read` va	Guest OS reads virtual address va	$\emptyset$ if va is Stealth `read` va otherwise
`read_hyper` va	The hypervisor reads virtual address va	$\emptyset$ if va is Stealth `read_hyper` va otherwise
`write` va val	Guest OS writes value val in va	$\emptyset$ if va is Stealth `write` va otherwise
`write_hyper` va val	The hypervisor writes value val in va	$\emptyset$ if va is Stealth `write` va otherwise
`new` va pa	Hypervisor extends the non stealth memory of the active OS with $va \mapsto ma$	`new` va pa
`new_sm` $stealth_va$ pa	Hypervisor extends the stealth memory of the active OS with $stealth_va \mapsto ma$	$\emptyset$
`del` va	Hypervisor deletes mapping for va from the current memory mapping of the active OS	$\emptyset$ if va is Stealth `del` va otherwise
`switch` o	Hypervisor sets o to be the active OS	`switch` o
`lswitch` pa	Hypervisor changes the current memory mapping of the active OS to be pa	`lswitch` pa
`hcall` hc	An OS requires privileged service hc to be executed by the hypervisor	`hcall` hc
`ret_ctrl`	Returns the execution control to the hypervisor	`ret_ctrl`
`chmod`	The hypervisor gives the execution control to the active OS	`chmod`
`page_pin` pa t	The memory page that corresponds to pa is registered and classified with type t for the active OS	`page_pin` pa t
`page_unpin` pa	The memory page of the active OS that corresponds to pa is un-registered	`page_unpin` pa
`silent`	Represents the silent action (no effects on the system)	`silent`

Fig. 3. Actions and their effects

$$\frac{\begin{array}{c} aos_act = (aos, running) \quad os_accessible(va) \\ get_page_mem(s, va) = (ma, pg) \quad pg = (RW\ _, OS\ aos, b) \\ cache_add(cache, va, ma, pg) = (cache', (ma', pg')) \\ mem[ma' := pg'][ma := pg] = mem' \quad tlb[va := ma] = tlb' \end{array}}{s = (oss, aos_act, hyp, mem, cache, tlb) \xhookrightarrow{\texttt{read}\ va} (oss, aos_act, hyp, mem', cache', tlb')}$$

$$\frac{\begin{array}{c} aos_act = (aos, running) \quad os_accessible(va) \\ get_page_mem(s, va) = (ma, pg) \quad pg = (RW\ _, OS\ aos, b) \\ cache_add(cache, va, ma, pg) = (cache', \bot) \quad tlb[va := ma] = tlb' \end{array}}{s = (oss, aos_act, hyp, mem, cache, tlb) \xhookrightarrow{\texttt{read}\ va} (oss, aos_act, hyp, mem', cache', tlb')}$$

Precondition

The action `read` *va* requires that the active OS *aos* is running, that *va* is accessible by *aos*, and that the current page table of *aos* maps the virtual address *va* to a machine address *ma* and a page *pg*. Moreover, *pg* is readable/writable.

Postcondition

After the execution of the action, the cache is updated with the entry associated to the pair (va, ma). The TLB is updated with the pair (va, ma).

If an entry is evicted from the cache (first rule), then this entry is written to the corresponding address in the new memory. Otherwise (second rule), the memory does not change.

Action `read_hyper` *va*

The hypervisor reads virtual address *va*

Rule

$$\frac{\begin{array}{c} aos_act = (aos, waiting) \quad get_page_mem(s, va) = (ma, pg) \quad pg = (RW\ _, OS\ aos, b) \\ cache_add(cache, va, ma, pg) = (cache', (ma', pg')) \\ mem[ma' := pg'][ma := pg] = mem' \quad tlb[va := ma] = tlb' \end{array}}{s = (oss, aos_act, hyp, mem, cache, tlb) \xhookrightarrow{\texttt{read_hyper}\ va} (oss, aos_act, hyp, mem', cache', tlb')}$$

$$\frac{\begin{array}{c} aos_act = (aos, waiting) \quad get_page_mem(s, va) = (ma, pg) \quad pg = (RW\ _, OS\ aos, b) \\ cache_add(cache, va, ma, pg) = (cache', \bot) \quad tlb[va := ma] = tlb' \end{array}}{s = (oss, aos_act, hyp, mem, cache, tlb) \xhookrightarrow{\texttt{read_hyper}\ va} (oss, aos_act, hyp, mem', cache', tlb')}$$

Precondition

The action `read_hyper` *va* requires that the active OS *aos* is in waiting mode (or equivalently, that the hypervisor is running). The current page table of the active OS *aos* maps the virtual address *va* to a machine address *ma* and a page *pg*. Moreover, *pg* is readable/writable.

Postcondition

After the execution of the action, the cache is updated with the entry associated to the pair (va, ma). The TLB is updated with the pair (va, ma). If an entry is evicted from the cache (first rule), then this entry is written to the corresponding address in the new memory. Otherwise (second rule), the memory does not change.

Action `write` *va val*

Guest OS writes value *val* in *va*

Rule

$$\frac{aos_act = (aos, running) \quad get_page_mem(s, va) = (ma, pg) \quad pg = (RW\ _, OS\ aos, b) \quad cache_add(cache, va, ma, (RW\ val, OS\ aos, b)) = (cache', (ma', pg')) \quad mem[ma' := pg'][ma := (RW\ val, OS\ aos, b)]_{pol} = mem' \quad tlb[va := ma] = tlb'}{s = (oss, aos_act, hyp, mem, cache, tlb) \xrightarrow{\texttt{write}\ va\ val} (oss, aos_act, hyp, mem', cache', tlb')}$$

$$\frac{aos_act = (aos, running) \quad get_page_mem(s, va) = (ma, pg) \quad pg = (RW\ _, OS\ aos, b) \quad cache_add(cache, va, ma, (RW\ val, OS\ aos, b)) = (cache', \bot) \quad mem[ma := (RW\ val, OS\ aos, b)]]_{pol} = mem' \quad tlb[va := ma] = tlb'}{s = (oss, aos_act, hyp, mem, cache, tlb) \xrightarrow{\texttt{write}\ va\ val} (oss, aos_act, hyp, mem', cache', tlb')}$$

Precondition

The action `write` va val requires that the active OS aos is running. Furthermore, the virtual address va is mapped to a machine address ma and a readable/writable page pg in the current page table of the active OS (get_page_mem).

Postcondition

There are two rules for the `write` action, one in which an entry is evicted from the cache when the written page is added, and the other in which no entry is evicted. In both cases the resulting state differs in the value val of the page associated to the pair (va, ma) in the cache $cache$, and in the TLB tlb. If $cache_add$ returns an entry (ma', pg') that was evicted from the cache, the memory in ma' is updated with pg'. The final value in memory of the page in ma is dependent on the write policy in use ($mem[ma := page]_{pol}$ updates the page in ma with $page$ in write-through policies, and it leaves it unchanged in write-back ones).

Action `write_hyper` va val

The hypervisor writes value val in va

Rule

$$\frac{aos_act = (aos, waiting) \quad get_page_mem(s, va) = (ma, pg) \quad pg = (RW\ _, OS\ aos, b) \quad cache_add(cache, va, ma, (RW\ val, OS\ aos, b)) = (cache', (ma', pg')) \quad mem[ma' := pg'][ma := (RW\ val, OS\ aos, b)]_{pol} = mem' \quad tlb[va := ma] = tlb'}{s = (oss, aos_act, hyp, mem, cache, tlb) \xrightarrow{\texttt{write_hyper}\ va\ val} (oss, aos_act, hyp, mem', cache', tlb')}$$

$$\frac{aos_act = (aos, waiting) \quad get_page_mem(s, va) = (ma, pg) \quad pg = (RW\ _, OS\ aos, b) \quad cache_add(cache, va, ma, (RW\ val, OS\ aos, b)) = (cache', \bot) \quad mem[ma := (RW\ val, OS\ aos, b)]]_{pol} = mem' \quad tlb[va := ma] = tlb'}{s = (oss, aos_act, hyp, mem, cache, tlb) \xrightarrow{\texttt{write_hyper}\ va\ val} (oss, aos_act, hyp, mem', cache', tlb')}$$

Precondition

The action `write_hyper` va val requires that the hypervisor is running. Furthermore, the virtual address va is mapped to a machine address ma and a readable/writable page pg in the current page table of the active OS (get_page_mem).

Postcondition

The postcondition of this action is the same as the postcondition of the `write` action.

2.2 Page table updates

Action `new` *va pa*

Hypervisor extends the non stealth memory of the activew OS with $va \mapsto ma$

Rule

$$\frac{\begin{array}{c} aos_act = (aos, waiting) \quad os_accessible(va) \quad oss[aos] = (pa', New\ va\ pa) \\ get_page_hyp(s, aos, pa) = (ma, pg) \quad non_stealth_cache_line(va) \\ \neg memory_alias(mem, va, ma) \\ get_page_hyp(s, aos, pa') = (ma', cpt) \quad cpt[va := ma] = cpt' \\ oss[aos := (pa', None)] = oss' \quad mem[ma' := cpt'] = mem' \\ remove_cache_va(cache, cpt, va) = cache' \quad tlb[va := \bot] = tlb' \end{array}}{(oss, aos_act, hyp, mem, cache, tlb) \xrightarrow{\texttt{new}\ va\ pa} (oss', aos_act, hyp, mem', cache', tlb')}$$

$$\frac{\begin{array}{c} aos_act = (aos, waiting) \quad os_accessible(va) \quad oss[aos] = (pa', New\ va\ pa) \\ get_page_hyp(s, aos, pa) = (ma, pg) \quad non_stealth_cache_line(va) \\ memory_alias(mem, va, ma) \\ get_page_hyp(s, aos, pa') = (ma', cpt) \quad cpt[va := ma] = cpt' \\ oss[aos := (pa', None)] = oss' \\ pg = (t, o, b) \quad mem[ma := (t, o, false)][ma' := cpt'] = mem' \\ remove_cache_va(cache, cpt, va) = cache' \quad remove_cache_ma\ cache'\ ma = cache'' \\ tlb[va := \bot] = tlb' \end{array}}{s = (oss, aos_act, hyp, mem, cache, tlb) \xrightarrow{\texttt{new}\ va\ pa} (oss', aos_act, hyp, mem', cache'', tlb')}$$

Precondition

The action `new` *va pa* requires that the hypervisor is running, that *va* is accessible by the active OS, and that *aos* has a pending new hypercall. Furthermore, *pa* should be mapped to some machine address *ma* and page *pg*.

Postcondition

In the state after the execution the current page table *cpt* is updated with the mapping of *va* to *ma*. If *va* appeared mapped to to some machine address in the current page table, this entry is removed from the cache by the *remove_new_va* function, and the old cached page value is written to memory.

If the new address is an alias, in addition to the above, the cache entry with *ma* as its address is removed from the cache (and written to memory). Additionally, the page in *ma* is marked as non cacheable.

Furthermore, the mapping of *va* to *ma* is removed from the TLB.

Notes

In order to achieve non-leakage of stealth data, it is necessary that accesses to the *stealth_va* do not replace any entry in the cache. The hypervisor ensures that this is the case by enforcing the *exclusion* property: it only allows operating systems to allocate virtual addresses that are not in the same cache line set as *stealth_va*.

Action `new_sm` *stealth_va pa*

Hypervisor extends the stealth memory of the active OS with $stealth_va \mapsto ma$

Rule

$$\frac{\begin{array}{c} aos_act = (aos, waiting) \qquad oss[aos] = (pa', New\ stealth_va\ pa) \\ get_page_hyp(s, aos, pa) = (ma, pg) \qquad pg = (RW\ _, OS\ aos, true) \\ \neg memory_alias(mem, stealth_va, ma) \qquad get_page_hyp(s, aos, pa') = (ma', cpt) \qquad cpt[stealth_va] = \bot \\ oss[aos := (pa', None)] = oss' \qquad cpt[stealth_va := ma] = cpt' \qquad mem[ma' := cpt'] = mem' \\ cache_add(cache, stealth_va, ma, pg) = (cache', _) \qquad tlb[stealth_va := ma] = tlb' \end{array}}{s = (oss, aos_act, hyp, mem, cache, tlb) \xhookrightarrow{\texttt{new_sm}\ stealth_va\ \ pa} (oss', aos_act, hyp, mem', cache', tlb')}$$

Precondition

The action `new_sm` *stealth_va pa* requires that the active OS *aos* is waiting for the hypervisor to extend its current page table *cpt* with *stealth_va*. The physical address *pa* maps to the machine address *ma* and page *pg* in the hypervisor mapping of *aos* (*get_page_hyp*). This page *pg* must be readable/writable and cacheable. Also, no page table can map a virtual address to *ma* (no *memory_alias*), and *stealth_va* is not mapped in *cpt*. This is needed in order to guarantee that the stealth page *pg* in *ma* is always cached and that no aliased pages are cached.

Postcondition

In the resulting state, the pending hypercall of *aos* is removed. The current page table *cpt* and *tlb* are updated with the mapping of *stealth_va* to *ma*. Furthermore, the new stealth page is immediately stored in *cache*.

Notes

The allocation of stealth pages requires that the new *stealth_va* is not aliased, in order to ensure that it is cacheable.

Action `del` *va*

Hypervisor deletes mapping for *va* from the current memory mapping of the active OS

Rule

$$\frac{\begin{array}{c} aos_act = (aos, waiting) \qquad os_accessible(va) \qquad oss[aos] = (pa', Del\ va) \\ get_page_hyp(s, aos, pa') = (ma', cpt) \\ oss[aos := (pa', None)] = oss' \qquad cpt[va := \bot] = cpt' \qquad mem[ma' := cpt'] = mem' \\ remove_cache_va(cache, cpt, va) = cache' \qquad tlb[va := \bot] = tlb' \end{array}}{s = (oss, aos_act, hyp, mem, cache, tlb) \xhookrightarrow{\texttt{del}\ va} (oss', aos_act, hyp, mem', cache', tlb')}$$

Precondition

The action `del` *va* requires that the hypervisor is running, that *va* is accessible by the active OS *aos*, and that *aos* has a pending del hypercall. Furthermore, *va* should be mapped to some machine address *ma* in the current page table of the active OS.

Postcondition

In the resulting state, the current page table *cpt* of *aos* is updated by deleting the mapping of *va* to *ma*. Additionally, if there exists a page in the cache for *va* in the current page table of *aos*, it is eliminated. Similarly, the mapping of *va* to *ma* is removed from the TLB.

Notes

The semantics of the `del` action is the same if the deleted virtual address is stealth or not. In both cases, the entry is removed from the cache and the mapping from the current page table of the operating system.

2.3 Context switches

Action `switch` o

Hypervisor sets o to be the active OS

Rule

$$\frac{\begin{array}{c} aos_act = (aos, waiting) \qquad oss[o] = (pa, None) \qquad get_page_hyp(s, o, pa) = (_, cpt) \\ (o, waiting) = aos_act' \qquad stealth_save(mem, cache) = mem' \\ stealth_add(stealth_drop(cache), mem, o, cpt) = cache' \qquad tlb_flush(tlb) = tlb' \end{array}}{(oss, aos_act, hyp, mem, cache, tlb) \xhookrightarrow{\texttt{switch}\ o} (oss, aos_act', hyp, mem', cache', tlb')}$$

Precondition

The context `switch` o action requires that the hypervisor is running, and that there is no hypercall pending for the OS o.

Postcondition

In the resulting state, o is the new active OS, its stealth pages, if any, are cached after saving the stealth cache pages into (stealth) memory, and the TLB tlb is (fully) flushed–the effect of tlb_flush is to return an empty TLB.

Notes

The semantics of this action reflects a distinguishing characteristic of VIPT caches, i.e. the cache is not flushed on context switches. As a consequence, the execution of one process of an OS may result in the eviction from the cache of the entries belonging to another concurrent OS; this behaviour can be exploited by an attacker to gain information on the execution of a victim OS. The fact that the stealth page of the active OS is always cached is what guarantees that these attacks cannot be performed on stealth pages.

Action `lswitch` pa

Hypervisor changes the current memory mapping of the active OS to be pa

Rule

$$\frac{\begin{array}{c} aos_act = (aos, waiting) \qquad oss[aos] = (pa', LSwitch\ pa) \\ get_page_hyp(s, aos, pa) = (ma, pg) \qquad pg = (PT\ _, OS\ aos, _) \qquad get_page_hyp(s, aos, pa') = (_, cpt) \\ oss[aos := (pa, None)] = oss' \qquad stealth_save(mem, cache) = mem' \\ stealth_add(stealth_drop(cache), mem, aos, cpt) = cache' \qquad tlb_flush(tlb) = tlb' \end{array}}{s = (oss, aos_act, hyp, mem, cache, tlb) \xhookrightarrow{\texttt{lswitch}\ pa} (oss', aos_act, hyp, mem', cache', tlb')}$$

Precondition

The action `lswitch` pa requires that the hypervisor is running, and that the active OS must be waiting for an hypercall to change its current memory mapping to be pa. Additionally, there exists a machine address ma, associated by the hypervisor mapping to the physical address pa, which is the address of a PT page owned by the active OS.

Postcondition

The effect of this action is to set pa as the address of the current page table of the active OS. Furthermore, its stealth page, if any, is cached after saving the stealth cache page into (stealth) memory. The TLB tlb is (fully) flushed.

2.4 Hypercall

Action `hcall` *hc*

The active OS requires privileged service *hc* to be executed by the hypervisor.

Rule

$$\frac{aos_act = (aos, running) \qquad oss[aos] = (pa, _) \qquad oss[aos := (pa, hc)] = oss' \qquad (aos, waiting) = aos_act'}{(oss, aos_act, hyp, mem, cache, tlb) \xhookrightarrow{\texttt{hcall}\ hc} (oss', aos_act', hyp, mem, cache, tlb)}$$

Precondition

The action `hcall` *hc* requires that the active OS is running.

Postcondition

In the resulting state, the active OS requires the execution of *hc* to the hypervisor, which takes control of the execution.

Notes

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.

2.5 Changes of execution mode

Action `ret_ctrl`

The active OS returns the execution control to the hypervisor.

Rule

$$\frac{aos_act = (aos, runnig) \qquad (aos, waiting) = aos_act'}{(oss, aos_act, hyp, mem, cache, tlb) \xhookrightarrow{\texttt{ret_ctrl}} (oss, aos_act', hyp, mem, cache, tlb)}$$

Precondition

The action `ret_ctrl` requires that the active OS is running.

Postcondition

In the resulting state, the hypervisor is running.

Action `chmod`

The hypervisor gives to the active OS the execution control.

Rule

$$\frac{aos_act = (aos, waiting) \qquad oss[aos] = (_, None) \qquad (aos, running) = aos_act'}{(oss, aos_act, hyp, mem, cache, tlb) \xhookrightarrow{\texttt{chmod}} (oss, aos_act', hyp, mem, cache, tlb)}$$

Precondition

The action `chmod` requires that the hypervisor is running, and that there is no hypercall pending for the active OS.

Postcondition

In the resulting state, the active OS is running.

2.6 Hypervisor mapping updates

Action `page_pin` $pa\ t$

The memory page that corresponds to physical address pa, for the active OS, is registered and classified with type t.

Rule

$$\frac{\begin{array}{c} aos_act = (aos, waiting) \qquad oss[aos] = (pa', Pin\ pa\ t) \\ hyp[aos][pa] = \bot \qquad free_madd(mem) = ma \\ oss[aos := (pa', None)] = oss' \qquad hyp[aos][pa := ma] = hyp' \\ mem[ma := (t, aos, true)] = mem' \end{array}}{(oss, aos_act, hyp, mem, cache, tlb) \xhookrightarrow{\texttt{page_pin}\ pa\ t} (oss', aos_act, hyp', mem', cache, tlb)}$$

Precondition

The action `page_pin` $pa\ t$ requires that the hypervisor is running, that the active OS must be waiting for an hypercall to *pin* the physical address pa of type t, and that pa must not be already allocated. In addition to that, there must be machine memory (ma) available.

Postcondition

The effect of the action is to create and allocate at machine address ma a new page of type t whose owner is the active OS and bind, in the hypervisor mapping, the physical address pa to ma. The rest of the state remains unchanged.

Action `page_unpin` pa

The memory page that corresponds to physical address pa, for the active OS, is un-registered.

Rule

$$\frac{\begin{array}{c} aos_act = (aos, waiting) \qquad oss[aos] = (pa', UnPin\ pa) \qquad pa' \neq pa \\ get_page_hyp(s, aos, pa) = (ma, pg) \qquad is_empty(pg) \qquad no_madd_in_PT(aos, ma, mem) \\ oss[aos := (pa', None)] = oss' \qquad hyp[aos][pa := \bot] = hyp' \\ mem[ma := (Other, No_Owner, true)] = mem' \end{array}}{s = (oss, aos_act, hyp, mem, cache, tlb) \xhookrightarrow{\texttt{unpin}\ pa} (oss', aos_act, hyp', mem', cache, tlb)}$$

Precondition

The action `page_unpin` pa requires that the hypervisor is running, that the active OS must be waiting for an hypercall to *unpin* the physical address pa, that pa is not the current page physical address of the active OS, and that a page associated with the machine address ma, mapped by pa in the hypervisor mapping, is registered in the memory. Additionally, if pg is of type PT, it must be empty, and ma can not appear in any page table of the active OS.

Postcondition

The action removes the mapping of pa to ma in the hypervisor mapping. In the resulting state, the page pg is un-registered –pg is released.

2.7 Silent

Action `silent`

Represents the silent action –the system does not advertise any effects.

Rule

$$\frac{}{(oss, aos_act, hyp, mem, cache, tlb) \xhookrightarrow{\texttt{silent}} (oss, aos_act, hyp, mem, cache, tlb)}$$

Precondition

This operation has no restrictions for execution.

Postcondition

The state does not change.

2.8 Invariance of valid state

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

Lemma 1 (Valid State Invariant).
$\forall\ (s\ s' : State)\ (a : Action)\ (o : os_ident),\ valid_state(s) \rightarrow\ s \xhookrightarrow{a} s'\ \rightarrow\ valid_state(s')$

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

3 Isolation

Isolation captures the idea that malicious guest operating systems cannot gain information about victim guest operating systems, and is formally stated as an indistinguishability property between platform executions that are suitably related. For concreteness, we consider a scenario with only two guest operating systems: a victim operating system o_v and an attacker operating system o_a.

3.1 Adversary model

We consider a very strong adversary that is able to observe the layout of the non-stealth cache (but not its contents), and the layout of the non-stealth memory.

We assume that the scheduler is adversarially controlled, i.e. at each step the adversary can decide whether the victim operating system or itself will execute. We further give the adversary the power to trigger the resolution of pending hypercalls. Formally, we model an adversary as a function $\mathcal{A}$ that takes a partial trace and returns either o_v, indicating that the victim operating system will execute next, or a next action of its choice.

3.2 Equivalence of states

We define an equivalence relation $\sim$ between states to model the partial view of the attacker operating system o_a on the state; thus, two states s and s' s.t. $s \sim s'$ coincide on all parts of the state exposed to the attacker. The fact that we allow dynamic memory allocation through the execution of the `pin` actions complicates the definition of this relation. Since we cannot state it directly in terms of machine addresses, we do it indirectly through the use of the hypervisor mapping physical addresses, which are the same in both executions. The definition of this relation follows:

Definition 1 (Equivalence of OS information). *The operating system information oss and oss′ are equivalent for the attacker, if the attacker has the same current page table, and the same hypercall in both states. Furthermore, the attacker must be either active in both states and have the same activity, or not active in the states.*

For the equivalence of the memory we consider two cases: the case a physical address is mapped to a readable/writable page (Equivalence of hypervisor mappings), and the case it is mapped to a page table (Equivalence of cache and memory mappings).

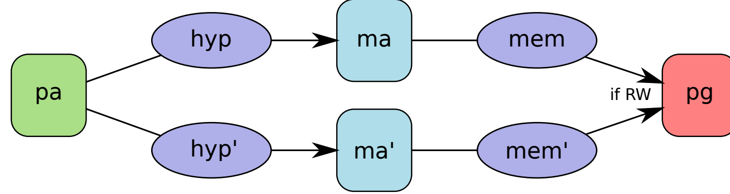

Fig. 4. Equivalence of hypervisor mappings

In the first (depicted in figure 4) we require that the attacker readable/writable pages are the same. Furthermore, the layout of the non-stealth memory pages of the victim must be the same (non stealth pages should have the same owner, and same cacheable flag, but arbitrary value).

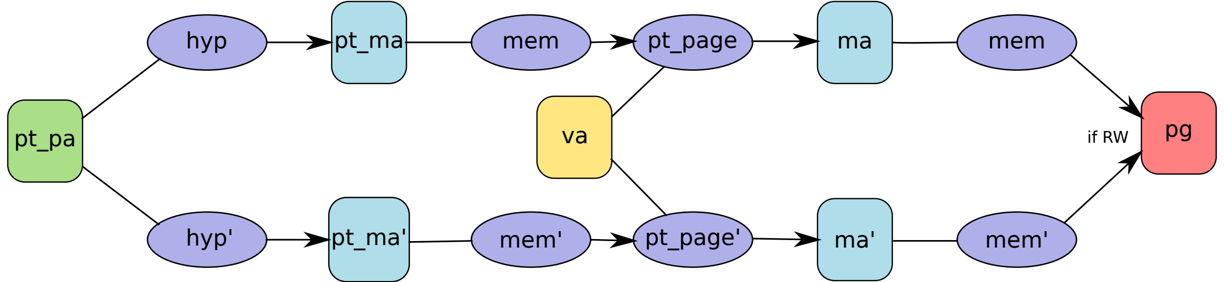

Fig. 5. Equivalence of cache and memory mappings

In the second, since machine addresses are arbitrary in both states, the page tables will be different. For them to be related, all entries in one page table must be in the other. Furthermore, readable/writable pages mapped by a virtual address should be the same, if the owner is the attacker, or have the same metadata if it is a non stealth victim page (figure 5).

More formally, these two equivalence relations are as follows:

Definition 2 (Equivalence of hypervisor mappings). *Two states s and s' have equivalent hypervisor mappings for the attacker ($s \sim^{hyp} s'$) if for every physical address pa, readable/writable page pg and machine address ma:*

- *if $get_page_hyp(s, o_a, pa) = (ma, pg)$, there exists ma' such that $get_page_hyp(s', o_a, pa) = (ma', pg)$;*
- *if $get_page_hyp(s, o_v, pa) = (ma, pg)$, and no page table maps stealth_va to ma, then there exists ma' such that $get_page_hyp(s', o_v, pa) = (ma', pg')$, where pg and pg' are equal except in their contents;*

and reciprocally for s'.

Definition 3 (Equivalence of cache and memory mappings). *Two states s and s' have equivalent cache and memory mappings for the attacker ($s \sim^{mem}$) if for every physical address pt_pa, page table pt_pg and machine address pt_ma, such that $get_page_hyp(s, o, pt_pa) = (pt_ma, pt_pg)$, there exists pt_ma' and pt_pg' such that $get_page_hyp(s', o, pt_pa) = (pt_ma', pt_pg')$ and for all virtual addresses va and machine address ma such that $pt_pg[va] = ma$ and $mem[ma]$ is readable/writable, then there exists ma' such that $pt_pg'[va] = ma'$ and moreover:*

- *if $o = o_a$ then $get_page_ma(s, ma) = get_page_ma(s, ma')$;*
- *if $o = o_v$ and va is not $stealth_va$, then $get_page_ma(s, ma)$ and $get_page_ma(s', ma')$ are equal except in their contents;*

and reciprocally for s'.

Definition 4 (Equivalence of cache histories). *Two cache histories history and history$'$ are equivalent for the attacker if for all non stealth cache index i $history[i] = history'[i]$.*

3.3 Unwinding lemmas

The equivalence relation $\sim$ is kept invariant by the execution of a victim stealth action. Furthermore, if the same attacker action or two non stealth victim actions with the same effect are executed in two equivalent states, the resulting states are also equivalent. These results are variations of standard non-interference unwinding lemmas (locally preserves and step-consistent unwinding lemmas) [3]:

Lemma 2 (Locally preserves unwinding lemma). *Let s and s' such that $s \xhookrightarrow{a,o_v} s'$ and $\mathsf{eff}(a) = \emptyset$. Then, $s \sim s'$.*

Lemma 3 (o_a step-consistent unwinding lemma). *Let s_1, s_2, s_1' and s_2' be states such that $s_1 \xhookrightarrow{a,o_a} s_2$, $s_1' \xhookrightarrow{a,o_a} s_2'$ and $s_1 \sim s_1'$. Then, $s_2 \sim s_2'$.*

Lemma 4 (o_v step-consistent unwinding lemma). *Let s_1, s_2, s_1' and s_2' be states such that $s_1 \xhookrightarrow{a,o_v} s_2$, $s_1' \xhookrightarrow{a',o_v} s_2'$, $eff(a) = eff(a')$ and $s_1 \sim s_1'$. Then, $s_2 \sim s_2'$.*

The proofs of these lemmas critically rely on the *inertia* property of cache [2]: upon adding a virtual address to the cache, the evicted virtual address, if any, is in the same cache line set as the added one; and on the *exclusion* property: the hypervisor ensures that guest operating systems can only allocate virtual addresses that are not in the same cache line set as the stealth virtual addresses.

3.4 Execution traces

The isolation property is eventually expressed on execution traces, rather than execution steps. 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 $s_0 \xhookrightarrow{a_0,o} s_1 \xhookrightarrow{a_1,o} s_2 \xhookrightarrow{a_2,o} s_3 \ldots$ such that every execution step $s_i \xhookrightarrow{a_i,o} s_{i+1}$ is valid. Formally, an execution trace is defined as a stream Θ of pairs of states and actions, such that for every $i \geq 0$, $s[i] \xhookrightarrow{a[i],o} s[i+1]$, where $\Theta[i].st = s[i]$, $t[i].act = a[i]$, $\Theta[i+1].st = s[i+1]$ and $\Theta[i+1].act = a[i+1]$.

We lift the indistinguishability relation to execution traces Θ and Θ' using the following co-inductive rules:

$$\frac{\mathsf{eff}(a) = \emptyset \quad \Theta \sim \Theta'}{s \xhookrightarrow{a,o_v} \Theta \sim \Theta'} \qquad \frac{\mathsf{eff}(a) = \emptyset \quad \Theta \sim \Theta'}{\Theta \sim s \xhookrightarrow{a,o_v} \Theta'} \qquad \frac{s \sim s' \quad \Theta \sim \Theta'}{s \xhookrightarrow{a,o_a} \Theta \sim s' \xhookrightarrow{a',o_a} \Theta'}$$

The property of isolation in which we are interested is a non-interference result on execution traces of the platform where the notion of indistinguishability amounts to equality of observations resulting from the execution of stealth actions. We show that no attacker observing a victim execution can gain knowledge from the behaviour induced by the execution of these kind of actions. More precisely, it can be proved that given

an operating system o_v and two execution traces Θ and Θ' such that in both traces the same non stealth actions are executed by o_v ($\Theta \approx \Theta'$), those traces are indistinguishable for any o_a that performs the same probing actions ($\Theta \sim \Theta'$). Notice that it must be required that the non stealth actions are the same in both execution traces because the execution of one such action, a `read` to a non stealth va in Θ for instance, may provoke a change in the cache that is otherwise unchanged if the action is not executed in Θ', leading to different observations for the attacker.

3.5 Interleaving

The main non leakage result on the model states that given two streams of victim actions executed in the same context (from equivalent initial states and the same attacker behaviour) the attacker cannot distinguish the two executions, provided both victim traces have the same sequence of non stealth actions.

We introduce the interleaving relation to characterize the merging of victim actions into a valid execution trace. As mentioned in Section 3.1, the attacker has control not only on the actions it executes (as we are modeling an adaptive attacker) but also on the scheduler, in the sense that it is the attacker who decides what operating system will execute next at any given point of the execution.

We model the attacker as a function that given the partial trace executed up to that moment, returns the next attacker action to execute, or handles control to the victim.

$$attacker_sched : partial_trace \to \{Victim \mid Attacker\ a\}$$

the result of this function is the same when two partial traces are equivalent with respect to the $\sim$ relation.

The interleaving relation makes use of this scheduler to merge the victim execution with the actions selected by the attacker:

$$\frac{attacker_sched(pt) = Victim \quad s \xhookrightarrow{a,o_v} \Theta[0].st \quad interleave(\Theta[0].st, pt \xhookrightarrow{a,o_a} s, v_str, \Theta)}{interleave(s, pt, a :: v_str, s \xhookrightarrow{a,o_a} \Theta)}$$

$$\frac{attacker_sched(pt) = Attacker\ a \quad s \xhookrightarrow{a,o_a} \Theta[0].st \quad interleave(\Theta[0].st, pt \xhookrightarrow{a,o_a} s, v_str, \Theta)}{interleave(s, pt, v_str, s \xhookrightarrow{a,o_a} \Theta)}$$

Given these definitions, we state our main non leakage theorem:

Theorem 1. *Let s_0 and s'_0 be two states such that $s_0 \sim s'_0$; v_str and v_str' two sequence of victim actions such that $v_str \approx v_str'$; and Θ and Θ' be traces such that $interleave(s_0, v_str, \Theta)$ and $interleave(s'_0, v_str', t')$. Then $\Theta \sim \Theta'$.*

This theorem provides a sufficient condition for non leakage of victim information that depends exclusively on the victim behaviour. This will allow us to connect this result with an application level analysis, to give formal guarantees of non leakage for programs (written in the C language) that make use of the StealthMem primitives. This connection is explained in more detail in the paper.

References

1. 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.
2. T. Kim, M. Peinado, and G. Mainar-Ruiz. Stealthmem: system-level protection against cache-based side channel attacks in the cloud. In *USENIX Security 2012*, pages 11–11, Berkeley, CA, USA, 2012. USENIX Association.
3. J. M. Rushby. Noninterference, Transitivity, and Channel-Control Security Policies. Technical Report CSL-92-02, SRI International, 1992.