INSTITUT FÜR INFORMATIK

DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Lehr- und Forschungseinheit für theoretische Informatik

Bachelor-Thesis

in Computer Science

Noninterference in the take-grant model for the seL4 microkernel

Andrea Kuchar

Advisor: Dr Martin Hofmann, PD Dr Ulrich Schöpp

Submission Date: 07-27-2018

Declaration of authorship

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references.

Munich, the 07-27-2018

....

Andrea Kuchar

Abstract

The thesis investigates the question if the specification of the seL4 access control system is strong enough to imply the Noninterference property. Using the verification of the Take-Grant-Protection Model [2] I deduce from it the Unwinding Theorem conditions of the nondeterministic intransitive Noninterference Model [1]. As the specifications and proofs of the take-grant model is developed in the theorem proof assistant Isabelle/HOL I use the same to verify the implication.

List of Figures

Contents

1 Introduction

1.1 Motivation

Nowadays our society becomes increasingly depended on computer systems. In more and more areas small computers take over the control. If it's our SmartTV, car or the control of the lights in our home. We are forced to confront our self with the topic how secure and reliable these systems are.
Especially if we entrust a computer our live this gets an essential meaning. We want to expect from board-computers in planes or cars that they are free from defects and not so easily hackable. This is not the reality. We know about cars whose board-computers simple can be taken over with a smartphone in the car next to it.
A key component in developing secure systems is the operating-system (OS) kernel of the system. The kernel has full access to hardware resources. One defect in the kernel can have the consequence that the security and reliability of the entire system can be lost.
The weakness of most previous kernels were their huge amount of code and mostly their monolithic design. This makes it impossible to review or verify the code. The weak point of the monolithic design is that not only fundamental functions as Interprocesscommunication, scheduling or memory management are implemented in the kernel mode but also functions like driver for hardware or virtual filesystems are integrated in it. This makes the system more vulnerable for bugs. One crashed modul can lead to a crash of the entire system.
The motivation behind microkernels is to reduce the possibility of bugs in the kernel code through reducing the kernelcode to an amount as minimal as possible and excluding functions from the kernel mode. With less code it becomes more feasible to guarantee the absence of defects within the kernel through formal verification.
Due to the fact that we feed our smartphones, tablets, board-computers, ... with growing amounts of sensitive informations like bank data, passwords, e-mails, chats, the significance of Security in the area of embedded systems increases.
Through isolation of small subsystems, like it is done in microkernels, the security already can be raised to a higher level. With testing one can depict an huge amount of bugs. But as Dijkstra said "Testing can only show the presence, not the absence, of bugs." [8]
Like I already mentioned less lines of codes make it more feasible to verify it relating to its specification. The seL4 microkernel is the first microkernel whose correctness is formally verified. It's a high-assurance, high-performance microkernel, primarily developed, maintained and formally verified by NICTA (now Trustworthy Systems Group at Data61) for secure embedded systems. It's security model is based on the take-grant model, which was extended for being able to reason about kernel memory consumption of components.

1.2 Aim of the thesis

With this thesis I want to show or disprove that the extended take-grant model is strong enough to show the noninterference property on it.
The security property of noninterference ensures that there is no unwanted information flow within a system. The take-grant model is an access control model. Therefore its duty is to "control" the access or the transfer of access on objects of a system. The noninterference property assured that there is no way information can flow to undesirable parties. Also with information about third parties that have access.
The thesis should investigate the different systemoperations of the model regarding the thereby occurring information flow.
With the collected information I want to decide two questions. First if the noninterference property can be verified with the existing model and second if the noninterference property is fulfilled.

1.3 Structure of the thesis

First I want to give an overview of the seL4 kernel, his set-up, the implementation of services and the memory management. For a better comprehension I then give a brief overview of the take-grant and the noninterference model. In chapter 3 the formalisation of the take-grant model takes place and in chapter 4 it's the formalisation of the noninterference model.
From chapter 5 on I dedicate myself to the validation of the noninterference property. In chapter 7 the validation is subdivided into the different systemoperations. To show the property for the model I have to extend the model in chapter 6.
Finally I'll take a short resume and give a prospect on the current status of the seL4 project and the possibilities to enhance this topic.

2 Requirements

2.1 The seL4 Microkernel

The seL4 [6] ist a small operation system kernel. It's based on the in the 1990s developed L4 microkernel and provieds a minimal number of services to applications, such as abstractions for virutal address spaces, threads, inter process comunication (IPC).
Each abstraction ist implemented by an kernel object with methodes dependent on the abstraction it supplies. The objects can be named and accessed by capablities which are also stored in kernel objects called *CNodes*.
Each capability contains an target object and potentially several access rights. The access rights can be `Read, Write, Grant` and `Create`. By invoking a capability that points to the kernel object with an corresponding method name, applications can invoke system calls. As arguments these system calls can have data or other capabilities.

2.1.1 System Calls

Kernel provided system calls:

- `send()`: The system call argument ist delivered to the target object and the application is allowed to continue. If the target is not able to receive and/or process the arguments immediately, the sending application will be blocked until the arguments can be delivered.
- `NBSend()`: Like `send()`. Exception: If the message is not deliverable it's silently droped.
- `Call()`: Like `send()` but the application is blocked until the object provieds a response, or the receiving application replies.
  If tthe argument is delivered to an application via Endpoint the receiver needs the right to respond to the sender. So in this case an additional capability is added to the arguments.
- `Wait()`: If the target object is not ready `Wait()` is used by an application to block until the object is ready.
- `Reply()`: Used to respond to a `Call()`, using the capability generated by the `Call()` operation.
- `ReplyWait()`: As a combination of `Reply()` and `Wait()` it's efficent for the common case that replying to a request and waiting for the next can be performed in a single system call.

2.1.2 Kernel Objects

The kernel implements several obejects to allocate the system operations [6].

- **CNodes**
  The capabilities to invoke system calls are stored in ***CNodes***. When created they get a fixed numer of slots that can be empty or contain a capability. The kernel conducts a **Capability Derivation Tree** (CDT) to keep records about the created capabilities and their associations. This is required for the revoke operation.
  They have the following operations:

  – `Mint()`
    creates a copy of an existing capability. The new capability is placed in a cpecified CNode slot and may have less rights than the parent capability. In the CDT the capability is placed as child of the original one.
  – `Copy()`
    is similar to the Mint operation. But the new capability has the same rights as the original one and in the CDT it's represented as a sibling of it.
  – `Move()`
    can maneuver a capability between two specified slots.
  – `Mutate()`
    moves the capability similar to `Move()` and is able to reduce it's rights like it's done in `Mint()` without an orignal copy remaining.
  – `Rotate()`
    moves two capabilities between three slots. Like two `Move()` operations.
  – `Delete()`
    can remove a capability from a specified slot.
  – `Revoke()`
    is used to remove a complete part of the CDT. From a defined capability on al children from the capability in the CDT are removed with `Delete()`.
  – `Recycle()`
    revokes all outstanding capabilities and reconfigures teh object to its initial state. So the object can be reused in for another purpose.

- **IPC Endpoints**
  Endpoints are used for the *interprocess communication* between threads. They can be devided into **synchronous (EP)** and **asynchronous (AEP)** endpoints. Threads in the seL4 kernel are grouped into security domains. Interprocess communication between different domains is only realised via AEPs. Generally capabilities to endpoints can be restricted to be read - or write - only.

- **TCP**
  A thread of execution in seL4 is represented by a *thread control block*. It's allways associated with a CSpace (provides the capabilities required to manipulate the kernel objects) and a VSpace (provides the virtual memory environment required to contain the code and data application).
  The TCB object has the following methods:
  `CopyRegisters()`, `ReadRegisters()`, `WriteRegisters()`, `SetPriority()`, `SetIPCBuffer()`, `SetSpace()`, `Configure()`, `Suspend()`, `Resume()`

- **Virtual Memory**
  Objects in the *virtual address space* (VSpace) implement services for the management of virtual memory which largely directly correspond to those of the hardware:
  Page Directory, Page Table, Page, ASID Control, ASID Pool
  Figure 1 showes how they are connected.

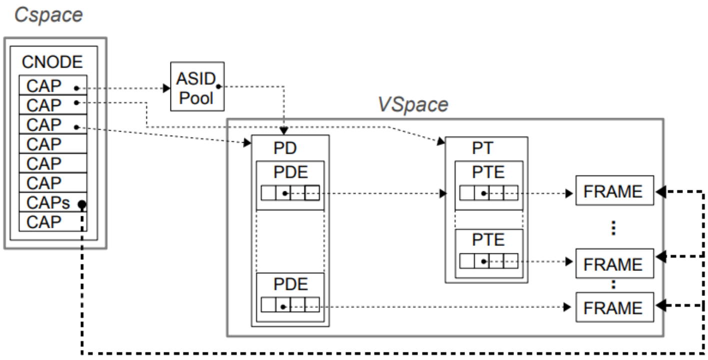

Figure 1: Internal representation of an application in seL4 [3]

- **Interrupt Objects**
  Device driver applications require `Interrupt Ojects` to be capable of receiving and acknowledging interrupts from hardware devices.

- **Untyped Memory**
  `Untyped memory objects (UMO)` seclude a fixed-sized, size-aligned,continuous region of the physical memory. Each object can be devided into a group of smaller untyped memory objects. With `Retype()` a number of new kernel objects created. It also returns capabilities to the new objects if it succeeds.

2.1.3 Memory Allocation Model

Important for the seL4 is that all kernel objects must be fully contributed for by capabilities.
At boot time the kernel pre-allocates all the memory required for the kernel to run. This includes the space for kernel code, data and kernel stack. The ressource manager has full authority over the untyped memory (UM) objects, generated by deviding the remain memory into these objects.
A capability to untyped memory can be refined into child capabilities, smaller sized untyped memory blocks or other kernel objects with the retype operation on UM objects.
The creator of an kernel object has full authority over the object. This "full authority" depends on the the object type.
Figure 2 shows a sample system architecture in wich a resource manager running at user-level has the authority to the remaining untyped memory after boot strapping.

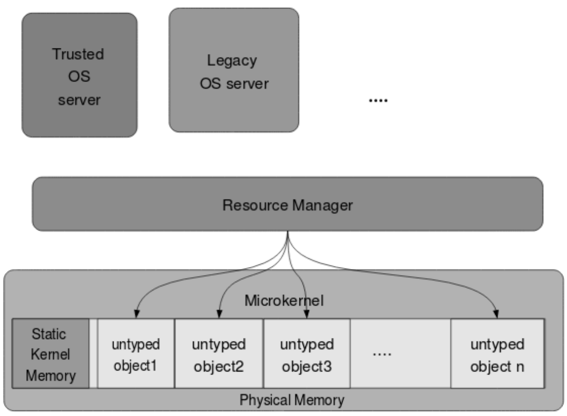

Figure 2: Sample System Configuration [2]

2.2 The Take-Grant Model

Protection or Acces control models specify, analyse and implemente secureity policies. The classical Take-Grant Model primary brought in by Lipton and Snyder, 1977 in "A Linear Time Algorithm for Deciding Subject Security".

2.2.1 The classical Model

The Take-Grant Model [2] represents the system as a directed graph where nodes represent subjects or objects in the system and arcs represent authority.
There are graph mutation rlues that represent the system operations that modify the autority distribution. The most common rules in the classical model are *take, grant, create* and *remove.*

- **take rule**: Let S,X,Y be three distinct vertices in the protection graph with an arc, labelled with α, from X to Y and one labelled with γ from S to X, such that t $\in \gamma$.

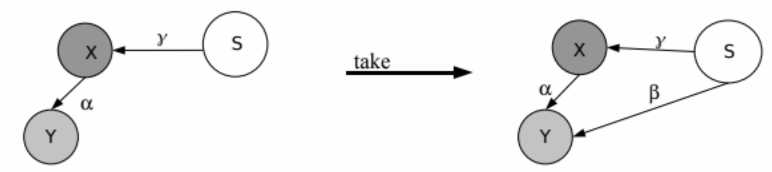

Figure 3: *Take* adds an edge from S to Y with the label $\beta \subseteq \alpha$. [2]

- **grant rule**: Let S,X,Y agein be three distinct vertices in the graph with an arc, labelled with α, from S to Y and one labelled with γ from S to X, such that g $\in \gamma$.

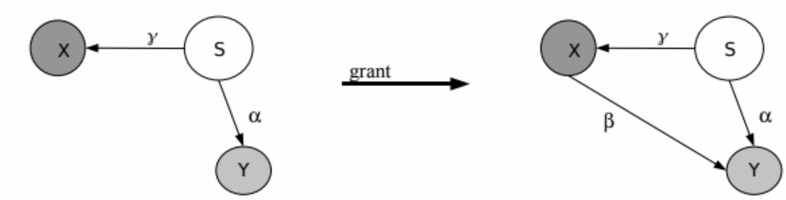

Figure 4: *Grant* adds an edge from X to Y with the label $\beta \subseteq \alpha$. [2]

- **create rule**: Let S be a vertex in the graph.

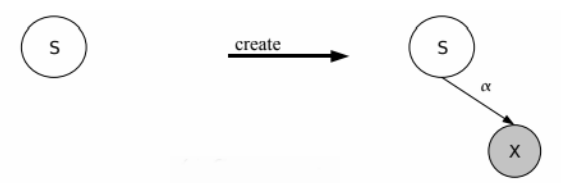

Figure 5: *Create* adds a new node X and an arc from S to X, labelled with α. [2]

- **remove rule**: Let S, X be vertices in the graph with an arc from S to X, labelled with α.

2.2.2 Take-Grant specified for the seL4

The Take-Grant Model specified in the paper "Noninterference for Operating System Kernels" [2] is a variant of the classical Take-Grant model.
The modification of the *create rule* is the most important one. In the kernel untyped capabilities transfer the authority that has to bei allocated and by the modification adding

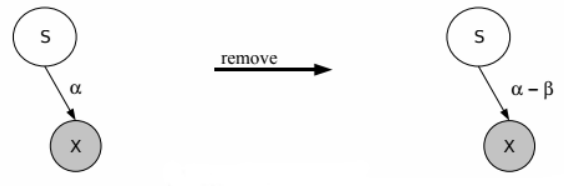

Figure 6: *Remove* deletes β labels from α or the arc itself if $\alpha - \beta = \{\}$. [2]

a new node to the protection graph corresponds to allocation a new object in the concrete kernel. So the only way to apply the create rule is if there is an outgoing arc with *create* authority. The *create* authority is represented by the label *c*.
Also the *remove rule* was modified. It doesn't remove parts of labels. Insted it removes the whole capability, which is the complete arc.
To diminish authority a capability has to be removed and newly created with diminished authority.
The kernel offers an operation called *revoke* wich removes a set of capabilities by mulitple applications of remove.
The goal of the paper "Noninterference for Operating System Kernels" was to show that it is accomblishable to implement isolated subsystems using the mechanisms of the seL4 kernel. [2]
An isolated sybsystem is an collection of connected *entities* enclosed in such a way that authority can neither get in nor out.
The exact specificaton of subsystems and entities follows in Chapter 3.

2.3 Noninterference

Noninterference is an enhancement of the information flow model, first published by Goguen and Meseguer in 1982 and updated in 1984. It ensures that objects and subject from different security levels don't interfere with those at other levels.
I use the noninterference formulation of Geoffrey Smith [7]. It says "Program c satisfies noninterference if, for any memories μ and ν that agree on L variables, the memories produced by running c on μ and on ν also agree on L variables (provided that both runs terminate successfully)."
This means if in a program two states ar equivalent on a low level domain The used noninterference formultaion for OS kernels [1] expands von Oheimb's notion of *noninfluence* [4].
The system is devided in different *domains*. An information flow *policy* $\leadsto$ specifies the allowed information flows between the domains: $u \leadsto v$ if information is allowed to flow from domain u to domain v.
For OS kernels we need an intransitive variant of noninterference, for wich $\leadsto$ can be intransitive.
The traditional Noninterference formulation was enhanced in in two ways:

1. Traditional formulations presume a static mapping `dom` from actions to domains. In an OS Kernel the mapping does not only depend on the actions but also on the current system state. So in the used formulation of Noninterference [1] `dom` also deppendes on the present state *s*.
   `dom` *a s* equates the domain associated with some action *a* that occurs from state *s*.

2. Due to the fact that the noninterference formulation in "Noninterference for Operating System Kernels" [1] was preserved by refinement, it is necessary to avert all *domain-visible* nondeterminisms.
   Domain-visible nondeterminism is nondeterminism that can be observed by any domain.

From every confidential source of information which is present in the refinement, such nondeterminisms can be abstracted. From this would result the existence of insecure refinements.
Lemma 2 [1] determine the restriction of no domain-visible nondeterminisms formally and will be clarified later.

3 Formalisation of the Take-Grant Model

3.1 Capabilities

In the Take-Grant model for seL4 [2] the authors waived the usual differentation betwenn subjects and objects and called all kernel objects *entities*.

The entities memory address identifies them and is modeled as a natural number.

```
type_synonym entity_id = nat
```

With each capability a set of rights is associated. There ar four access rights in the system model:

```
datatype rights = Read | Write | Grant | Create
```

- *Read* authorises the reading of information from another entity.
- *Write* authorises the writing of information to another entity.
- *Grant* authorises the passing of a capability to another entity.
- *Create* authorises the creation of new entities, which models the behavior of untyped memory objects.

A capability has two fields:

1. An identifier which names an target-entity
2. A set of rights which defines which system-operations the source-entity is authorisied to perform on the target-entity.

```
record  cap =  entity :: entity_id
               rights :: rights set
```

An entity has a set of capabilities:

```
record  entity = caps :: cap set
```

The systems state includes two flields:

1. The `heap`, which stores the entities of the system like an arry form address `0` up to and excluding `next_id`.
2. `next_id` contains slot for next entity without overlapping with an existing one.

```
record  state =  heap :: entity_id ⇒ entity
                 next_id :: entity_id
```

3.2 System Operations

The system operations of the seL4 are determined in the data type `sysOps`.

```
datatype   sysOps  =  SysNoOp entity_id
                   |  SysRead entity_id cap
                   |  SysWrite entity_id cap
                   |  SysCreate entity_id cap cap
                   |  SysGrant entity_id cap cap rights set
                   |  SysRemove entity_id cap cap
                   |  SysRevoke entity_id cap
```

The entity_id in each operation is the entity initiating the operation. The first named capability is the one that is being invoked. The second capability for `SysCreate` points to the target entity for the new capability. For `SysGrant` it's the passed capability and for `SysRemove` it's the one that has to be removed. The rights set in `SysGrant` necessary for the initiating entity to have the option only to transport a subset of the authority it offers to the receiver.
The `diminish` function applies this mask on the given acces rights:

```
diminish ::  "cap ⇒ rights set ⇒ cap" where
diminish c R ≡ c⦇rights := rights c ∩ R⦈
```

`legal` defines on what terms any system operation is allowed.

```
legal ::  "sysOPs ⇒ state ⇒ bool" where

    "legal  (SysNoOp e) s          =  isEntityOf s e"
  | "legal  (SysCreate e c₁ c₂) s  =  (isEntityOf s e ∧ c₁, c₂ ⊆ caps_of s e ∧
                                      Grant ∈ rights c₂ ∧ Create ∈ rights c₂)"
  | "legal  (SysRead e c) s        =  (isEntityOf s e ∧ c ∈ caps_of s e ∧ Read
                                      ∈ rights c)"
  | "legal  (SysWrite e c) s       =  (isEntityOf s e ∧ c ∈ caps_of s e ∧ Write
                                      ∈ rights c)"
  | "legal  (SysGrant e c₁ c₂ r) s =  (isEntityOf s e ∧ isEntityOf s (entity c₁)
                                      ∧ c₁,c₂ ⊆ caps_of s e ∧ Grant ∈ rights c₁)"
  | "legal  (SysRemove e c₁ c₂) s  =  (isEntityOf s e ∧ c₁ ∈ caps_of s e)"
  | "legal  (SysRevoke e c) s      =  isEntityOf s e ∧ c ∈ caps_of s e"
```

`isEntityOf` tests the existence of an `entity_id`, `caps_of` issues the set of all capabilities contained in the entity with the address `r` in state `s`.

The original executions of `SysRead` and `SysWrite` don't have an underlying function. For implying the noninterference property I have to include what happens if an entity reads or writes a value from another entity. For this purpose I defined a `readOperation` and a `writeOperation`.
The `step'` and `step` functions define the execution of a single system operation:

```
step' ::  "sysOPs ⇒ state ⇒ state" where
    "step'  (SysNoOp e) s          =  s"
  | "step'  (SysRead e c) s        =  readOperation e c s"
  | "step'  (SysWrite e c) s       =  writeOperation e c s"
  | "step'  (SysCreat e c₁ c₂) s   =  createOperation e c₁ c₂ s"
  | "step'  (SysGrant e c₁ c₂ R) s =  grantOperation e c₁ c₂ R s"
  | "step'  (SysRemove e c₁ c₂) s  =  removeOperation e c₁ c₂ s"
  | "step'  (SysRevoke e c) s      =  revokeOperation e c s"

step ::  "sysOps ⇒ state ⇒ state" where
step cmd s ≡ if legal cmd s then step' cmd s else s
```

The new defined functions `readOperation` and `writeOperation`:

```
readOperation :: "entity_id ⇒ cap ⇒ modify_state" where
"readOperation e c s ≡ s⦇ heap := (heap s)(e := ⦇caps = caps_of s e, eValue = value_of s (entity
c)⦈)⦈"

writeOperation :: "entity_id ⇒ cap ⇒ modify_state" where
"writeOperation e c s ≡ s⦇ heap := (heap s)(entity c := ⦇caps = caps_of s (entity c), eValue
= value_of s e|))|)"
```

The rest of the system operation stay as they are:

```
createOperation :: "entity_id ⇒ cap ⇒ cap ⇒ modify_state" where
createOperation e c1 c2 s ≡
 let   nullEntity = ⦇cap = , eValue = NULL⦈ ;
       newCap = ⦇entity = next_id s, rights = all_rights⦈;
       newTarget = ⦇caps = newCap  caps_of s (entity c2), eValue = NULL⦈
 in    s⦇heap := (heap s)(entity c2 := newTarget, next_id s := nullEntity), next_id := next_id s+1⦈"


grantOperation :: "entity_id ⇒ cap ⇒ cap ⇒ rights set ⇒ modify_state" where
"grantOperation e c1 c2 R s ≡
s⦇heap := (heap s)(entity c1 := ⦇caps = diminish c2 R ∪ caps_of s (entity c1), eValue = value_of
s (entity c1)⦈)⦈"


removeOperation :: "entity_id ⇒ cap ⇒ cap ⇒ modify_state" where
"removeOperation c1 c2 s ≡ s⦇heap := (heap s)(entity c1 := ⦇caps = caps_of s (entity c1) - c2,
eValue = value_of s (entity c1)⦈)⦈"
```

4 Formalisation of the Noninterference Model

5 Validation of Noninterference

Confidentiality is one of the Noninterference Properties.

$$\texttt{confidentiality-u} \equiv \forall\ a\ d\ s\ t\ s'\ t'.\ \ \texttt{reachable}\ s \wedge \texttt{reachable}\ t \wedge s \overset{d}{\sim} t \wedge (\texttt{dom}\ a\ s \rightsquigarrow d \longrightarrow s \overset{dom\ a\ s}{\sim} t \wedge (s,s') \in \texttt{Step}\ a \wedge (t,t') \in \texttt{Step}\ a \longrightarrow s' \overset{d}{\sim} t'$$

To validate confidentiality for the take-grant model I had to define $\mathtt{s} \overset{\mathtt{d}}{\sim} \mathtt{t}$ for the model. $\mathtt{s} \overset{\mathtt{d}}{\sim} \mathtt{t}$ means that for every entity `e` reachable from an etnity in `d` the status of `e` in `s` and `t` has to be the same.
I named the function `aquiv_nonin`. It compares the value and capabilities of `e` and the entities of the subsystem `e` is located in for `s` and `t`.

`aquiv_nonin :: "state ⇒ state ⇒ subSysT ⇒ bool" where`
`"aquiv_nonin s t d ≡ ∀ e ∈ d. value"` First I tried to validate confidentiality for the different system operations as they are defined in the take-grant-model. With this model it's impossible to decide whether a change of value has been recognized by another domain.
In the paper an entity only include a set of capabilites. For my purpose I need the option to access the content of the entities. This ist because the rules for noninterference state that no information is allowed to flow from one domain to another. This includes the information stored in the kernel objects. Therefore I extendet the original record `entity` by adding a *value* modelled by a natural number.

My entity type:

```
record   entity =   caps ::  cap set
                    eValue ::  nat
```

After this change it was feasible to deside confidentiality for this model in the following way.

I took one Low-level-Subsystem and one High-level-Subsystem with entities in them and tested for different right-sets and different operations if the confidentiality-property holds. The following shows an example of this approach:

- e_1 ∈ H, e_2 ∈ L, c_1 ∈ s, c_2 ∈ t
- H equates a High level domain that implements the subsystem 'H'
- L equates a Low level domain that implements the subsystem 'L'

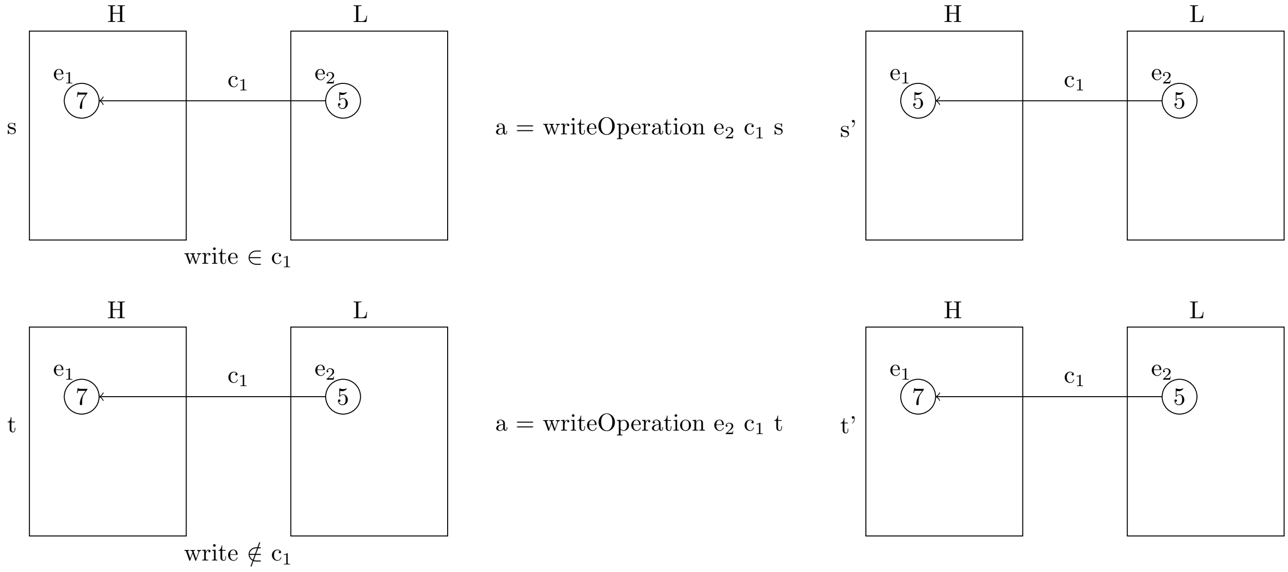

Figure 7: Confidentiality of Write 1

* $s \overset{L}{\sim} t \Rightarrow$ aquiv_nonin s t L
** writeOperation e_2 c_2 t changes e_1 $\in$ H not e $\in$ L
*** writeOperation e_2 c_1 s = s' $\overset{****}{=}$ s
**** legal(SysRead e_2 c_1) s = false

$\forall$ e$\in$L.

value_of s' e $\overset{***}{=}$ value_of s e $\overset{*}{=}$ value_of t e $\overset{**}{=}$ value_of t' e
$\wedge$ caps_of s' e $\overset{***}{=}$ caps_of s e $\overset{*}{=}$ caps_of t e $\overset{**}{=}$ caps_of t' e
$\wedge$ subSys s' e $\overset{***}{=}$ subSys s e $\overset{*}{=}$ subSys t e $\overset{**}{=}$ subSys t' e
$\Rightarrow$ aquiv_nonin s' t' L $\Rightarrow$ s' $\overset{L}{\sim}$ t'

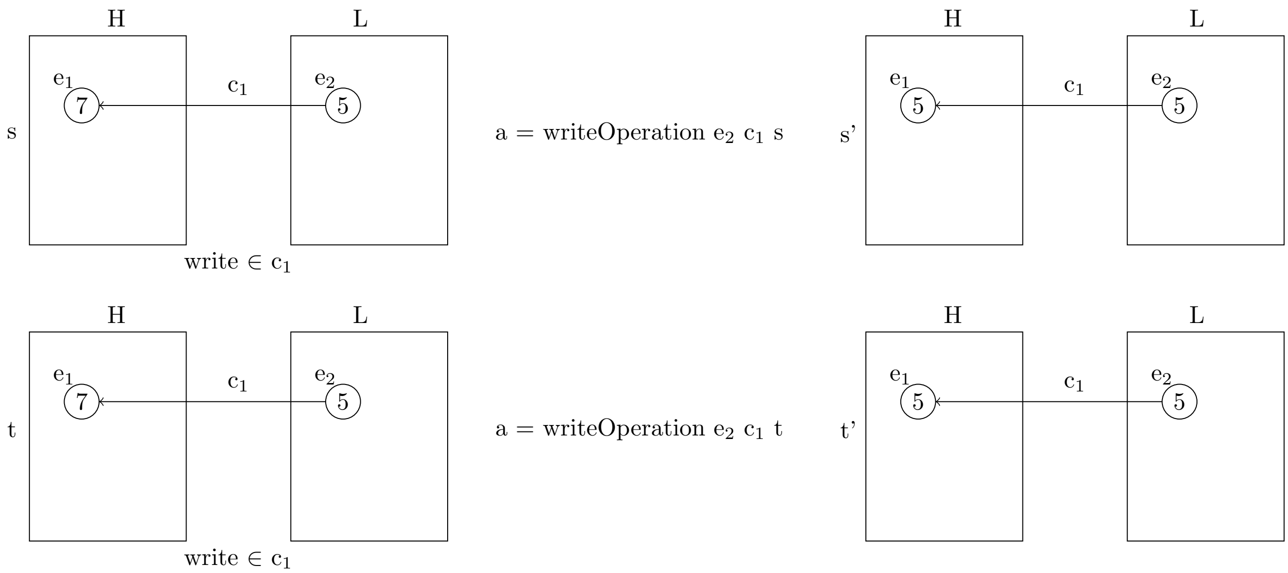

Figure 8: Confidentiality of Write 2

* $s \overset{L}{\sim} t \Rightarrow$ aquiv_nonin s t L

** writeOperation e_2 c_1 s changes e_1 ∈ H no e ∈ L
*** writeOperation e_2 c_2 t changes e_1 ∈ H no e ∈ L

∀ e∈L.

$$\begin{aligned}
& \text{value_of s' e} \overset{**}{=} \text{value_of s e} \overset{*}{=} \text{value_of t e} \overset{***}{=} \text{value_of t' e} \\
\wedge\ & \text{caps_of s' e} \overset{**}{=} \text{caps_of s e} \overset{*}{=} \text{caps_of t e} \overset{***}{=} \text{caps_of t' e} \\
\wedge\ & \text{subSys s' e} \overset{**}{=} \text{subSys s e} \overset{*}{=} \text{subSys t e} \overset{***}{=} \text{subSys t' e}
\end{aligned}$$

$\Rightarrow$ aquiv_nonin s' t' L $\Rightarrow$ s' $\overset{L}{\sim}$ t'

6 Redesign of the take-grant-model

This procedure worked until I came to the remove-operation. There I got the problem, that an entity in the given model is allowed to delete a capability and with that also an object in another domain without any restrictions:

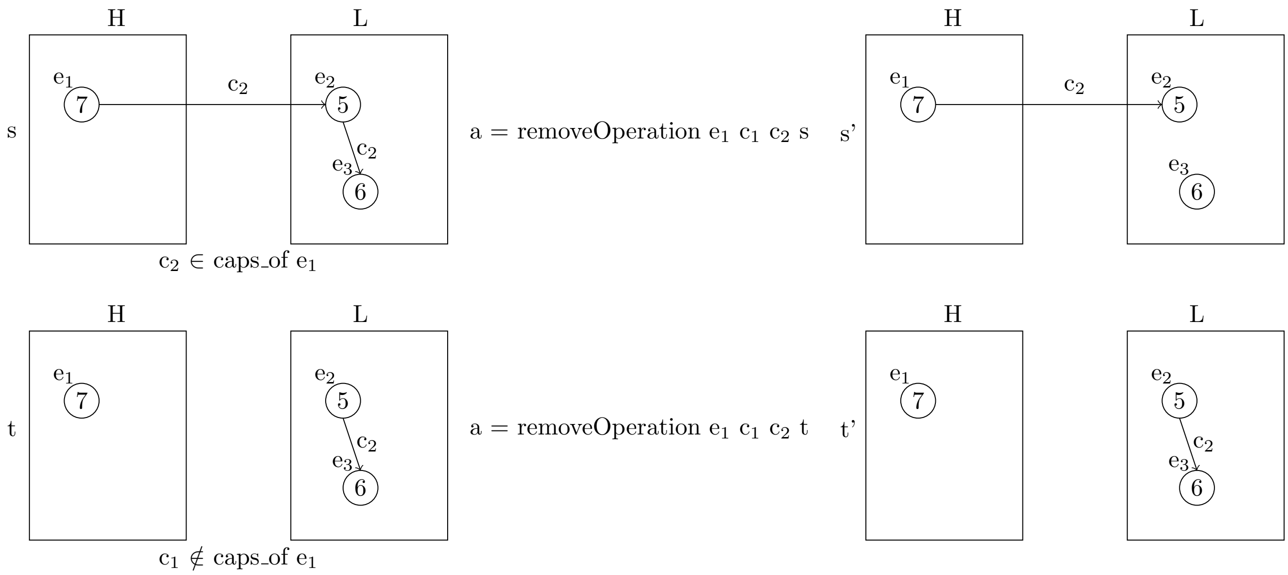

Figure 9: No confidentiality for Remove

To research into this problem I desided to classify the entities by their types,corresponding to the kernel specification [6]:

- Untyped
- TCB
- Synchronous IPC Endpoint (SEP)
- Asychronous IPC Endpoint (AEP)
- CNode
- VSpace
- Interrupt Controller
- Interrupt Handler

The following table showes the different object types with the different operation executable on them and the corresponding take- grant system calls:

Capability Type	Concrete Kernel	protection model
Untyped	Retype Revoke	sequence of *SysCreate* *SysRevoke*
TCB	TreadControl Exchange Registers Yield	*SysNoOP, SysGrant* *SystWrite* or *SysRead* *SysNoOP*
Synchronous IPC (Endpoint)	Send IPC Wait IPC Grant IPC	*SysWrite* or *SysNoOP* *SysRead* *SysWrite, SysGrant* or *SysNoOP*
Asynchronous IPC (AsyncEndpoint)	Send Event Wait Event	*SysWrite* *SysRead*
CNode	imitate mint Remove Revoke Move Recycle	*SysGrant* *SysGrant* *SysRemove* *SysRevoke* *SysGrant, SysRemove* *SysRevoke*, sequence of *SysRemove*
VSpace	Install Mapping Remove Mapping Remap initialise	*SysGrant* *SysRemove* *SysRemove, SysGrant* *SysNoOP*
Frame	-	-
InterruptController	Register interrupt Unregister interrupt	*SysGrant* *SysRemove*
Interrupt Handler	Acknowledge interrupt	*SysWrite*

Table 1: Relationship: operation of concrete kernel ⟷ of protection model [5]

To discem the different object types I need to revise the entity record and the preconditions for the different system operations.

New dataype for the object types:

```
datatype   eType  =  Untyped
                  |  TCB
                  |  SEP
                  |  AEP
                  |  CNode
                  |  VSpace
                  |  IContr
                  |  IHandl
```

The final version of the `entity` record:

```
record   entity =  caps ::  cap set
                   eValue ::  nat
                   eType ::  eType
```

The revised version of the `legal` function:

```
legal ::  "sysOPs ⇒ state ⇒ bool" where
```

```
  "legal  (SysNoOp e) s          =  isEntityOf s e"
| "legal  (SysCreate e c1 c2) s  =  (isEntityOf s e ∧ c1, c2 ⊆ caps_of s e ∧
                                    Grant ∈ rights c2 ∧ Create ∈ rights c2) ∧
                                    eType (entity c1 = Untyped"
| "legal  (SysRead e c) s        =  (isEntityOf s e ∧ c ∈ caps_of s e ∧ Read
                                    ∈ rights c) ∧ eType (entity c) = TCB ∨ SEP ∨ AEP"
| "legal  (SysWrite e c) s       =  (isEntityOf s e ∧ c ∈ caps_of s e ∧ Write
                                    ∈ rights c) ∧ eType (entity c) = TCB ∨ SEP ∨ AEP
                                    ∨ IHandl"
| "legal  (SysGrant e c1 c2 r) s =  (isEntityOf s e ∧ isEntityOf s (entity c1)
                                    ∧ c1,c2 ⊆ caps_of s e ∧ Grant ∈ rights c1) ∧
                                    eType (entity c1) = TCB ∨ SEP ∨ CNode ∨ VSpace ∨
                                    IContr"
| "legal  (SysRemove e c1 c2) s  =  (isEntityOf s e ∧ c1 ∈ caps_of s e) ∧
                                    eType (entity c1) = CNode ∨ VSpace ∨ IContr"
| "legal  (SysRevoke e c) s      =  isEntityOf s e ∧ c ∈ caps_of s e ∧
                                    eType (entity c) = Untyped ∨ CNode"
```

As mentioned in chapter 3.2 (System Operations) the step function first proves if a system operation is "legal" in state s. If it is the system operation is performed otherwise the new state s' is defined as s' = s. This means that if a system operation is not legal nothing happens. For the validation I took a subsystem (SS1) of one Domain (D1) and another subsystem (SS2) of a second Domain (D2).
In chapter 2.1.2 (Kernel Objects) I explained that the only communication between Domains goes through *Asynchronous Endpoints*.
The following picture shows an example of how the objects and methods can be placed in the domains and how the connection to *Asynchronous Endpoints* is implemented if the information is allowed to flow from Domain 1 to Domain 2.

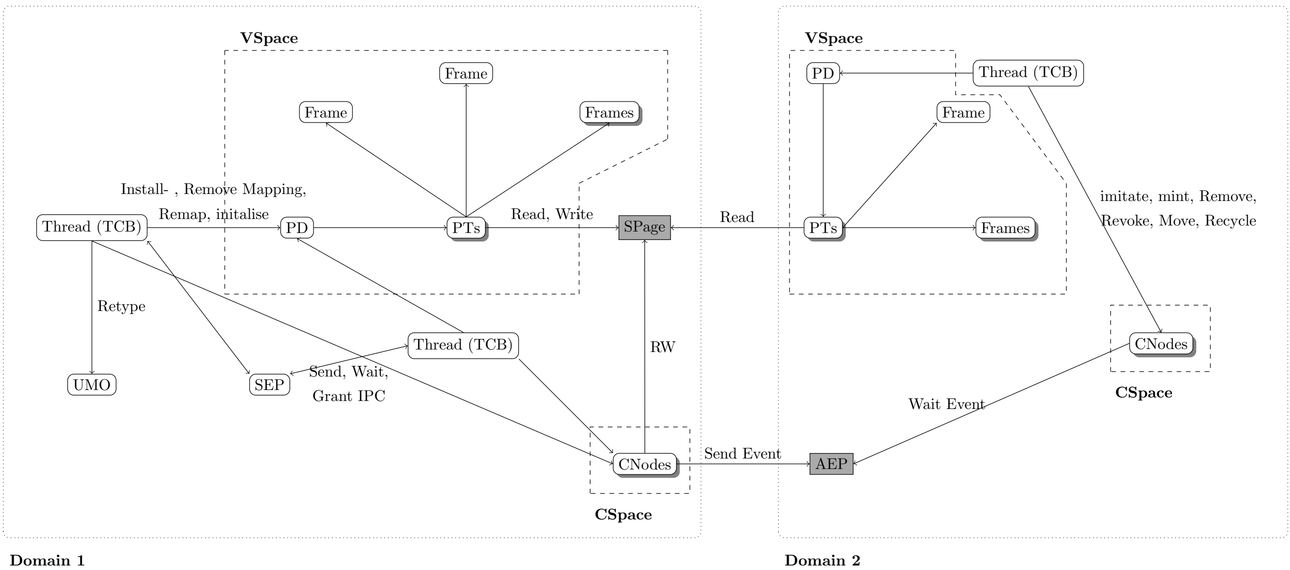

Figure 10: Objects and Methods in the kernel

7 Validation with the new model

I examine each operation of the protection model and distinguish therefore between the different object types.
I assume that information is allowed to flow from Domain 1 to Domain 2 but not from Domain 2 to Domain 1.

$\Rightarrow$ D1$\rightsquigarrow$D2 but D2$\not\rightsquigarrow$D1

Further I assume that state s is equivalent to state t for Domain 1.

$\Rightarrow$ s $\overset{D1}{\sim}$ t $\Rightarrow$ aquiv_nonin s t D1

In this chapter I show that the criteria for the equvalence relation still holds in Domain 1, between s' and t', after every type of operation.

7.1 Create

Create corresponds to the *Retype* operation on untyped memory. Each Domain has a own and fixed section of memory. So the UMO for *Retype* is located in the same Domain as the implementing entity. Also the created entity is placed in the same Domain as in the CDT it is a child of the UMO.

7.1.1 Create on UMO

The following picture shows how a create operation in one Domain changes or not changes the equivalence criteria in the other domain that is not allowed to get infomation from the primer one.

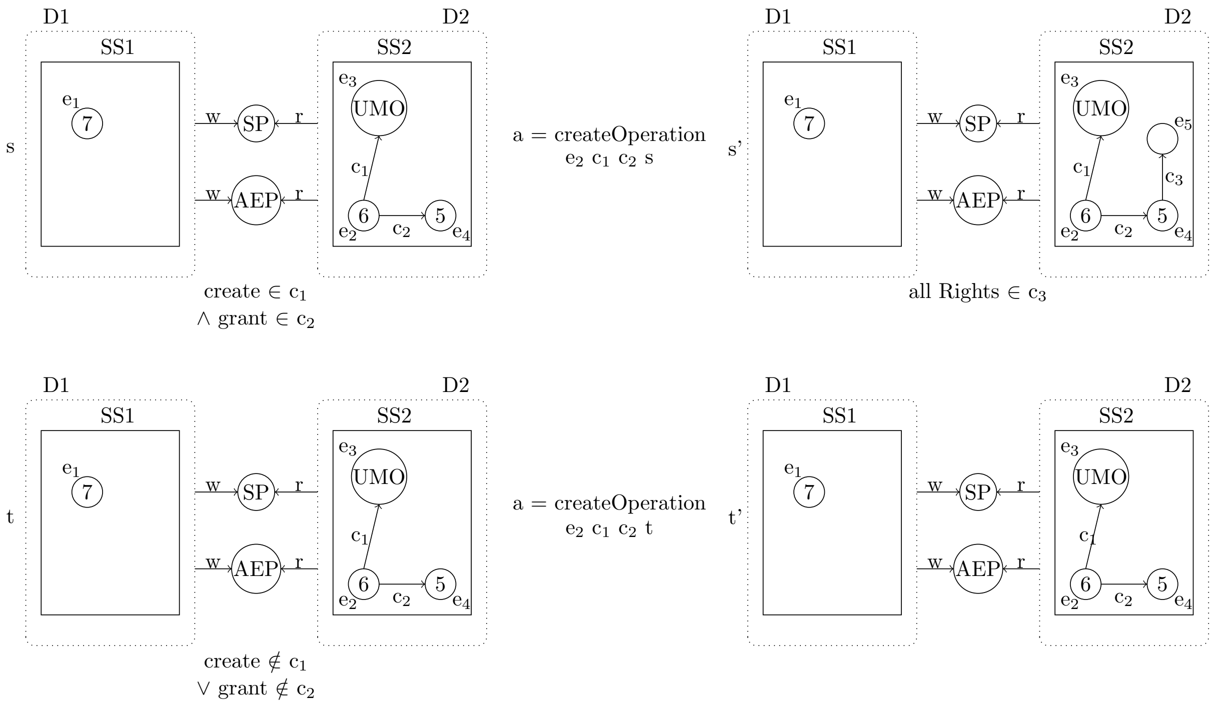

Figure 11: Noninterference for Create on Untyped Memory Objects

I have to show that if s $\overset{D1}{\sim}$ t and (s,s') $\in$ Step a and (t,t') $\in$ Step a then s' $\overset{D1}{\sim}$ t'. s $\overset{D1}{\sim}$ t was defined in Chapter 5 as the boolean fuction aquiv_nonin s t D1. The function is true if all entities e $\in$ D1 have the same value in s and t (value_of s e = value_of t e), if they also have the same capabilities in s and t (caps_of s e = caps_of t e) and if D1 has the same entities in s and t (subSys s e = subSys t e).
In the following section I show that value_of s' e = value_of t' e, caps_of s' e = caps_of t e and subSys s' e = subSys t' e for all e $\in$ D1 after the execution of createOperation e_2 c_1 c_2 s respectively createOperation e_2 c_1 c_2 t. And with that I show that aquiv_nonin s' t' D1. Then I can say from my definition that s' $\overset{D1}{\sim}$ t'.

Preconditions:

*	$s \overset{D1}{\sim} t \equiv$ `aquiv_nonin s t D1`
**	createOperation e_2 c_1 c_2 s creates $e_3 \in D2$ and does not change or create any $e \in D1$
***	legal (SysCreate e_2 c_1 c_2) t = false $\Rightarrow$ t' = t

Proof of the noninterference property for create on UMO: $\forall$ e $\in$ D1.

$$(\text{value\_of } s' \ e \overset{**}{=} \text{value\_of } s \ e \overset{*}{=} \text{value\_of } t \ e \overset{***}{=} \text{value\_of } t' \ e$$
$$\wedge \ \text{caps\_of } s' \ e \overset{**}{=} \text{caps\_of } s \ e \overset{*}{=} \text{caps\_of } t \ e \overset{***}{=} \text{caps\_of } t' \ e$$
$$\wedge \ \text{subSys } s' \ e \overset{**}{=} \text{subSys } s \ e \overset{*}{=} \text{subSys } t \ e \overset{***}{=} \text{subSys } t' \ e)$$
$$\Rightarrow \text{aquiv\_nonin } s' \ t' \ D1 \Rightarrow s' \overset{D1}{\sim} t'$$

With s' $\overset{D1}{\sim}$ t' the noninterference property for Create on an untyped memory object is fulfilled.

7.1.2 Create on all other object types inside a domain

If `create` is performed on another object type than an untyped memory object, the function step' (SysCreate e c_1 c_2)s does nothing.
The following figure showes the createOperation for every other object type inside a domain.

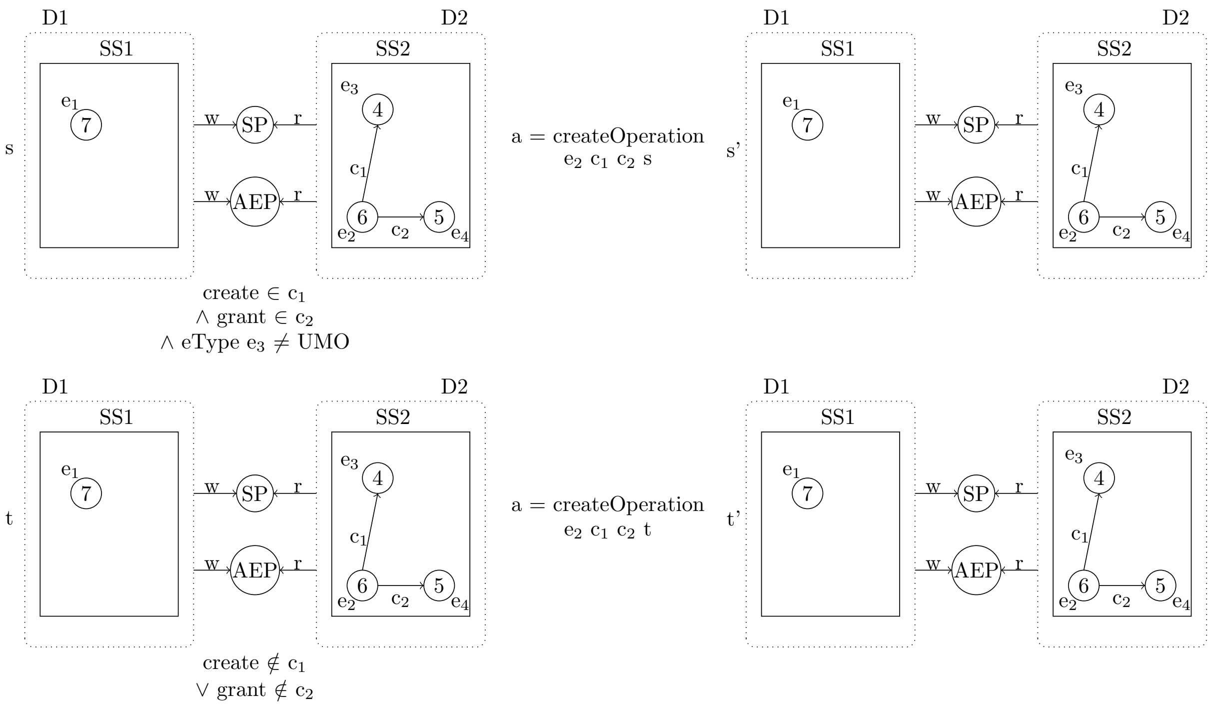

Figure 12: Noninterference for Create on object types $\neq$ Untyped Memory Objects

Preconditions:

*	$s \overset{D1}{\sim} t \equiv$ `aquiv_nonin s t D1`
**	legal (SysCreate e_2 c_1 c_2) s = false $\Rightarrow$ s' = s
***	legal (SysCreate e_2 c_1 c_2) t = false $\Rightarrow$ t' = t

Proof of the noninterference property for create on other object types in a domain: $\forall$ e $\in$ D1.

$$
\begin{array}{ll}
 & (\text{value\_of s' e} \overset{**}{=} \text{value\_of s e} \overset{*}{=} \text{value\_of t e} \overset{***}{=} \text{value\_of t' e} \\
\wedge & \text{caps\_of s' e} \overset{**}{=} \text{caps\_of s e} \overset{*}{=} \text{caps\_of t e} \overset{***}{=} \text{caps\_of t' e} \\
\wedge & \text{subSys s' e} \overset{**}{=} \text{subSys s e} \overset{*}{=} \text{subSys t e} \overset{***}{=} \text{subSys t' e})
\end{array}
$$

$\Rightarrow$ aquiv_nonin s' t' D1 $\Rightarrow$ s' $\overset{D1}{\sim}$ t'

With s' $\overset{D1}{\sim}$ t' the noninterference property for Create on other object types in a domain is fulfilled.

7.1.3 Create on Asynchronous IPC Endpoint objects

The next figure showes create on the AEP endpoints.

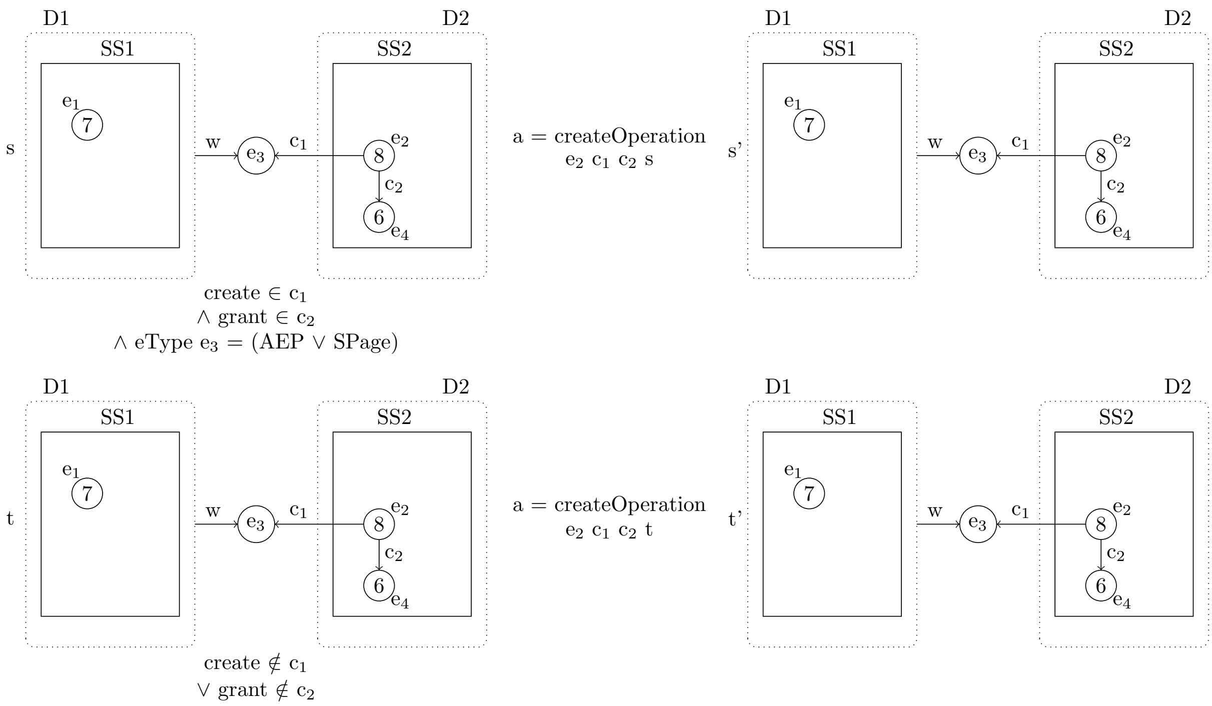

Figure 13: Noninterference for Create on object types = AEP

Preconditions:

*	s $\overset{D1}{\sim}$ t $\equiv$ `aquiv_nonin s t D1`
**	legal (SysCreate e_2 c_1 c_2) s = false $\Rightarrow$ s' = s
***	legal (SysCreate e_2 c_1 c_2) t = false $\Rightarrow$ t' = t

Proof of the noninterference property for create on Asynchronous IPC Endpoint objects

$\forall$ e $\in$ D1.

$$
\begin{array}{ll}
 & \text{value\_of s' e} \overset{**}{=} \text{value\_of s e} \overset{*}{=} \text{value\_of t e} \overset{***}{=} \text{value\_of t' e} \\
\wedge & \text{caps\_of s' e} \overset{**}{=} \text{caps\_of s e} \overset{*}{=} \text{caps\_of t e} \overset{***}{=} \text{caps\_of t' e} \\
\wedge & \text{subSys s' e} \overset{**}{=} \text{subSys s e} \overset{*}{=} \text{subSys t e} \overset{***}{=} \text{subSys t' e}
\end{array}
$$

$\Rightarrow$ aquiv_nonin s' t' D1 $\Rightarrow$ s' $\overset{D1}{\sim}$ t'

With s' $\overset{D1}{\sim}$ t' the noninterference property for Create on Asynchronous IPC Endpoint objects is fulfilled.

7.2 Grant

Grant operation can only be done inside a domain from a TCB, Synchronous IPC, CNode, VSpace or Interrupt Controller object. The only object types that are able to have contact to different domains are Async IPC objects.

7.2.1 Grant on TCB, SEP, CNode, VSpace or IContr objects

I show that any grant operation inside a Domain on one of the named objects does not affect the values, capabilities or entities of another domain.
Because it's the same for every of the given objects, in this model, I generalized e_4 = TCB ∨ SIPC ∨ CNode ∨ VSpace ∨ IContr.

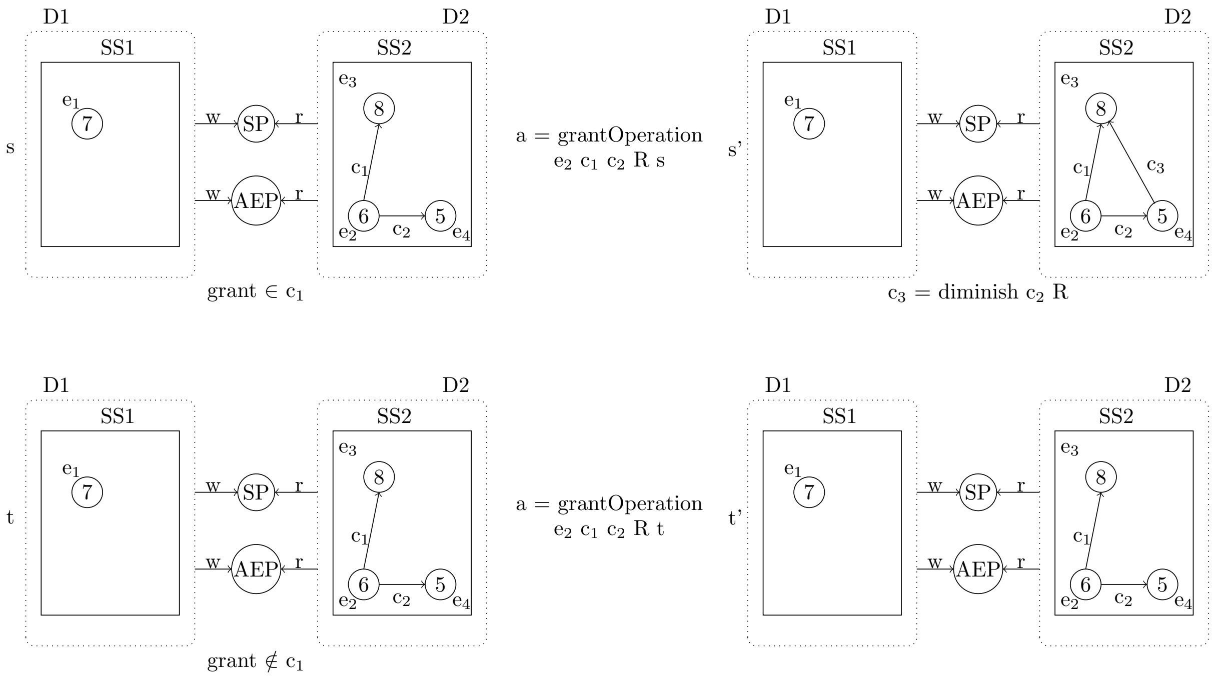

Figure 14: Noninterference for Grant on an TCB, Synchronous IPC Endpoint, CNode, VSpace or Interrupt Controller object

*	s $\overset{D1}{\sim}$ t ⇒ aquiv_nonin s t D1
**	grantOperation e_2 c_1 c_2 R s creates c_3 ∈ D2 and does not change or create any capability ∈ D1
***	legal (SysGrant e_2 c_1 c_2 R) t = false ⇒ t' = t

∀ e ∈ D1.

value_of s' e $\overset{**}{=}$ value_of s e $\overset{*}{=}$ value_of t e $\overset{***}{=}$ value_of t' e

∧ caps_of s' e $\overset{**}{=}$ caps_of s e $\overset{*}{=}$ caps_of t e $\overset{***}{=}$ caps_of t' e

∧ subSys s' e $\overset{**}{=}$ subSys s e $\overset{*}{=}$ subSys t e $\overset{***}{=}$ subSys t' e

⇒ aquiv_nonin s' t' D1 ⇒ s' $\overset{D1}{\sim}$ t'

7.2.2 Grant on other objects inside a domain

In this paragraph I show that an execution of the grant operation on an object other than TCB, SEP, CNode, VSpace, Interrupt Controller or the object type that establish a communication interface between domains: AEP.

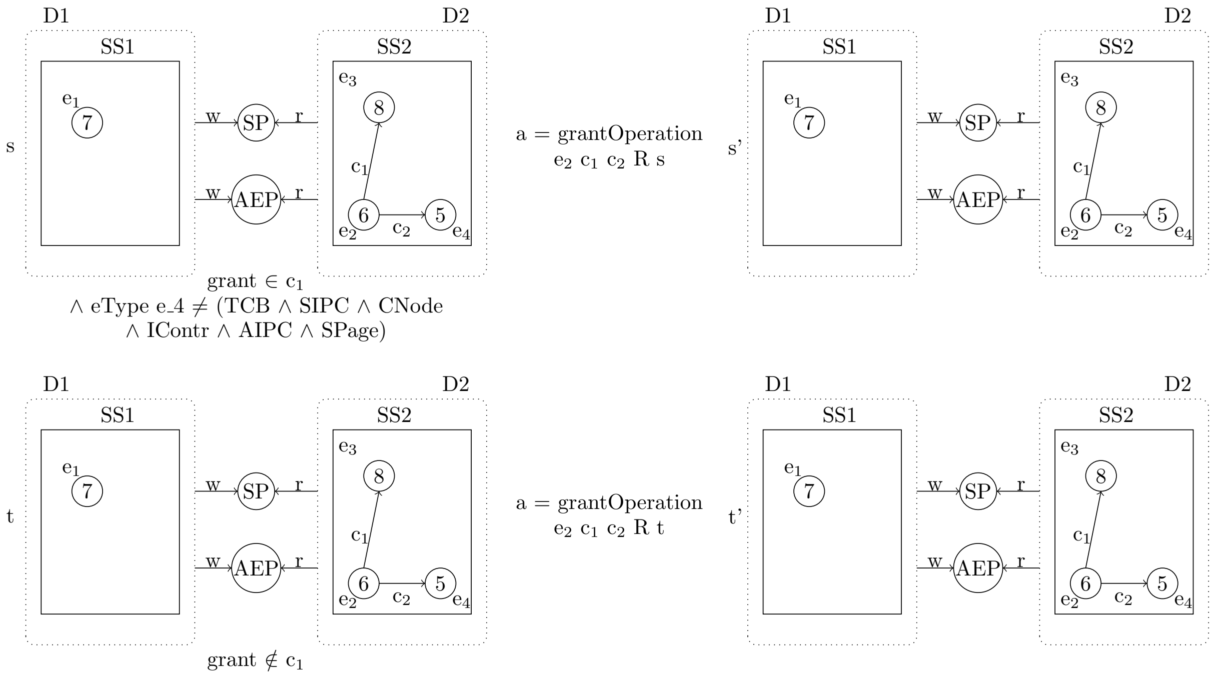

Figure 15: Noninterference for Grant on an object $\neq$ TCB, Synchronous IPC Endpoint, CNode, VSpace, Interrupt Controller object or AEP

*	s $\overset{D1}{\sim}$ t $\Rightarrow$ aquiv_nonin s t D1
**	legal (SysGrant e_2 c_1 c_2) s = false $\Rightarrow$ s' = s
***	legal (SysGrant e_2 c_1 c_2 R) t = false $\Rightarrow$ t' = t

$\forall$ e $\in$ D1.

$$\begin{array}{ll} & \text{value\_of s' e} \overset{**}{=} \text{value\_of s e} \overset{*}{=} \text{value\_of t e} \overset{***}{=} \text{value\_of t' e} \\ \wedge & \text{caps\_of s' e} \overset{**}{=} \text{caps\_of s e} \overset{*}{=} \text{caps\_of t e} \overset{***}{=} \text{caps\_of t' e} \\ \wedge & \text{subSys s' e} \overset{**}{=} \text{subSys s e} \overset{*}{=} \text{subSys t e} \overset{***}{=} \text{subSys t' e} \end{array}$$

$\Rightarrow$ aquiv_nonin s' t' D1 $\Rightarrow$ s' $\overset{D1}{\sim}$ t'

7.2.3 Grant on Asynchronous IPC Endpoint objects

The next picture illustrates grant on the two object types connecting different domains. In both cases the operation is not legal.

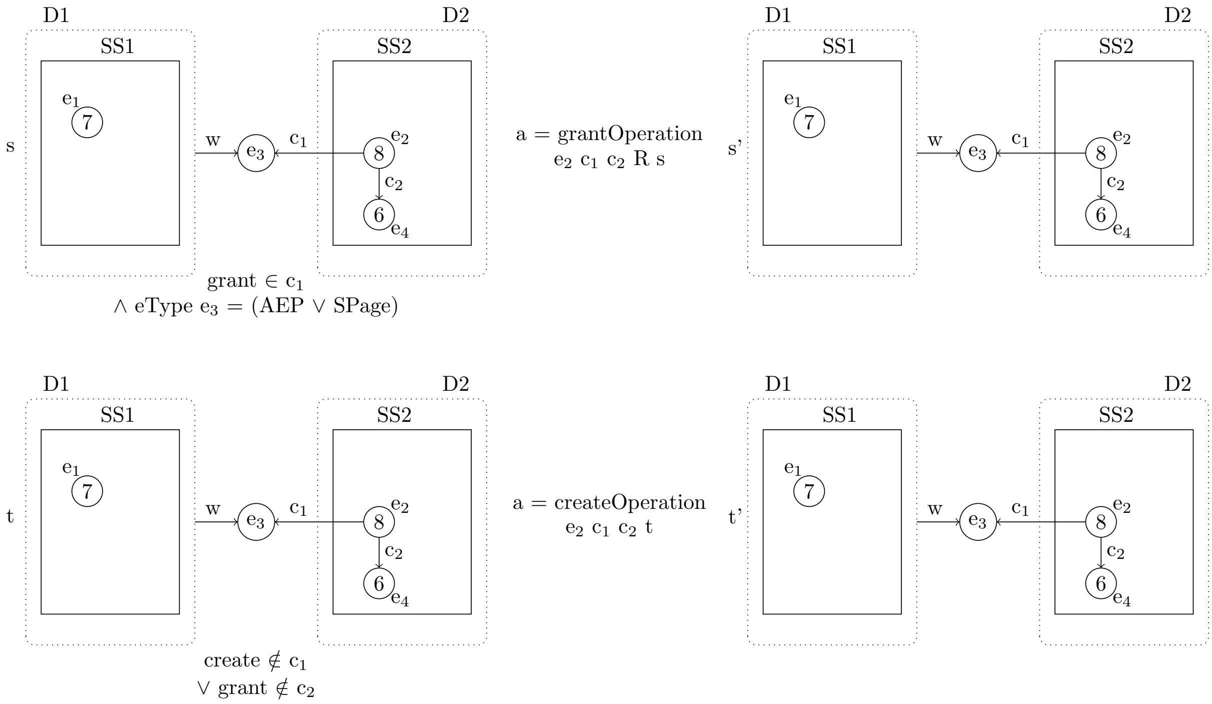

Figure 16: Noninterference for Grant on an Asychronous IPC Endpoint object

*	s $\overset{D1}{\sim}$ t ⇒ aquiv_nonin s t D1
**	legal (SysGrant e_2 c_1 c_2) s = false ⇒ s' = s
***	legal (SysGrant e_2 c_1 c_2) t = false ⇒ t' = t

∀ e ∈ D1.

$$\begin{aligned}
& \text{value_of s' e} \overset{**}{=} \text{value_of s e} \overset{*}{=} \text{value_of t e} \overset{***}{=} \text{value_of t' e} \\
\wedge\ & \text{caps_of s' e} \overset{**}{=} \text{caps_of s e} \overset{*}{=} \text{caps_of t e} \overset{***}{=} \text{caps_of t' e} \\
\wedge\ & \text{subSys s' e} \overset{**}{=} \text{subSys s e} \overset{*}{=} \text{subSys t e} \overset{***}{=} \text{subSys t' e}
\end{aligned}$$

⇒ aquiv_nonin s' t' D1 ⇒ s' $\overset{D1}{\sim}$ t'

7.3 Write

Write can be executed on TCB, SEP, AEP and Interrupt Handler objects.

7.3.1 Write on TCB, SEP or Interrupt Handler object

First I show the create operation on all executable objects inside a domain. So in the next figure e_3 = TCB ∨ SEP ∨ IHandl.

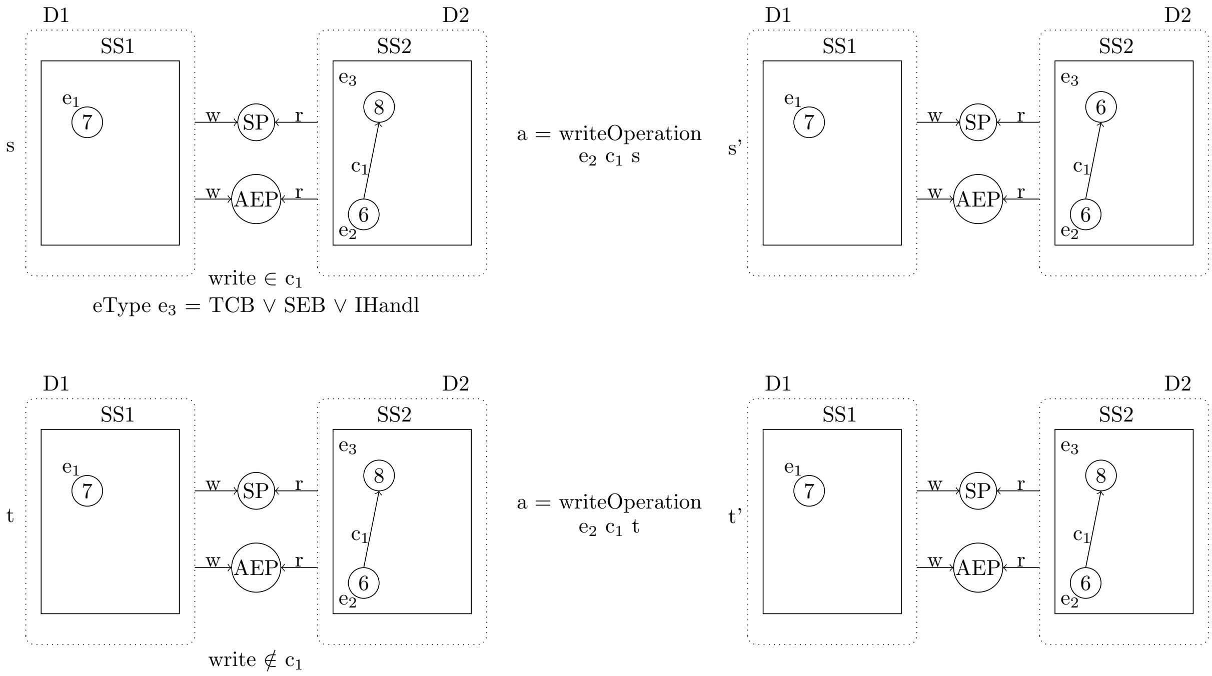

Figure 17: Noninterference for Write on a TCB, Sychronous IPC Endpoint or Interrupt Handler object

*	$s \overset{D1}{\sim} t \Rightarrow$ aquiv_nonin s t D1
**	writeOperation e_2 c_1 s only changes the value of an entity $\in$ D2 nothing in D1
***	legal (SysWrite e_2 c_1) t = false $\Rightarrow$ t' = t

$\forall$ e $\in$ D1.

$$\begin{aligned} & \text{value\_of s' e} \overset{**}{=} \text{value\_of s e} \overset{*}{=} \text{value\_of t e} \overset{***}{=} \text{value\_of t' e} \\ \wedge\ & \text{caps\_of s' e} \overset{**}{=} \text{caps\_of s e} \overset{*}{=} \text{caps\_of t e} \overset{***}{=} \text{caps\_of t' e} \\ \wedge\ & \text{subSys s' e} \overset{**}{=} \text{subSys s e} \overset{*}{=} \text{subSys t e} \overset{***}{=} \text{subSys t' e} \end{aligned}$$

$\Rightarrow$ aquiv_nonin s' t' D1 $\Rightarrow$ $s' \overset{D1}{\sim} t'$

7.3.2 Write on other objects ≠ TCB, SEP, IHandl and AEP

Like in 7.1 Create and 7.2 Grant there are other object types inside a domain, which are not executeable with the write operation. Those are CNodes, VSpaces, UMOs and Interrupt Controller.

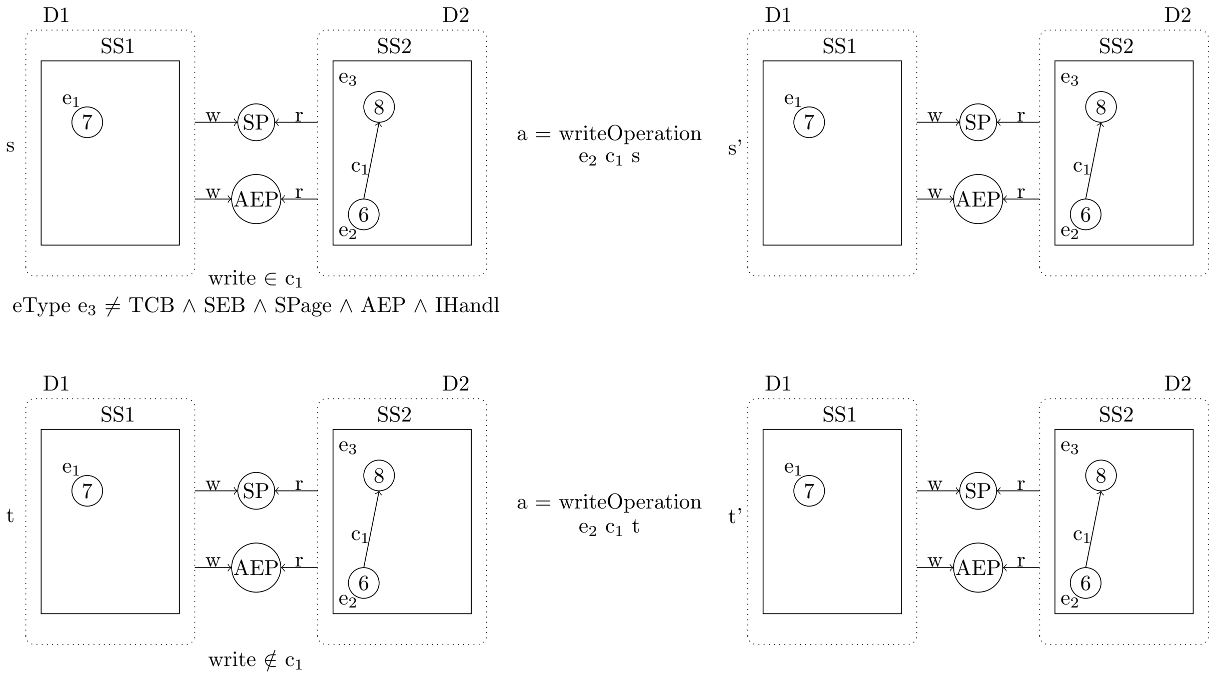

Figure 18: Noninterference for Write on other objects ≠ TCB, SEP, IHandl and AEP

*	$s \overset{D1}{\sim} t \Rightarrow$ aquiv_nonin s t D1
**	legal (SysWrite e_2 c_1) s = false $\Rightarrow$ s' = s
***	legal (SysWrite e_2 c_1) t = false $\Rightarrow$ t' = t

$\forall$ e $\in$ D1.

$$\text{value\_of s' e} \overset{**}{=} \text{value\_of s e} \overset{*}{=} \text{value\_of t e} \overset{***}{=} \text{value\_of t' e}$$

$$\wedge \quad \text{caps\_of s' e} \overset{**}{=} \text{caps\_of s e} \overset{*}{=} \text{caps\_of t e} \overset{***}{=} \text{caps\_of t' e}$$

$$\wedge \quad \text{subSys s' e} \overset{**}{=} \text{subSys s e} \overset{*}{=} \text{subSys t e} \overset{***}{=} \text{subSys t' e}$$

$\Rightarrow$ aquiv_nonin s' t' D1 $\Rightarrow s' \overset{D1}{\sim} t'$

7.3.3 Write on an AEP object from Domain 2

In Chapter 7 I defined the precondition $\Rightarrow$ D1$\leadsto$D2 but D2$\not\leadsto$D1. That means the rights from Domain 2 on Asychronous Endpoints are restricted to `read`. If the read operation is called from Domain 2 it looks like it is illustrated in Figure 19. The policy prescribes that information is only allowed to flow from Domain 1 to Domain 2 but not from Domain 2 to Domain 1. This has the consequence that `write` can not be part of c_1.

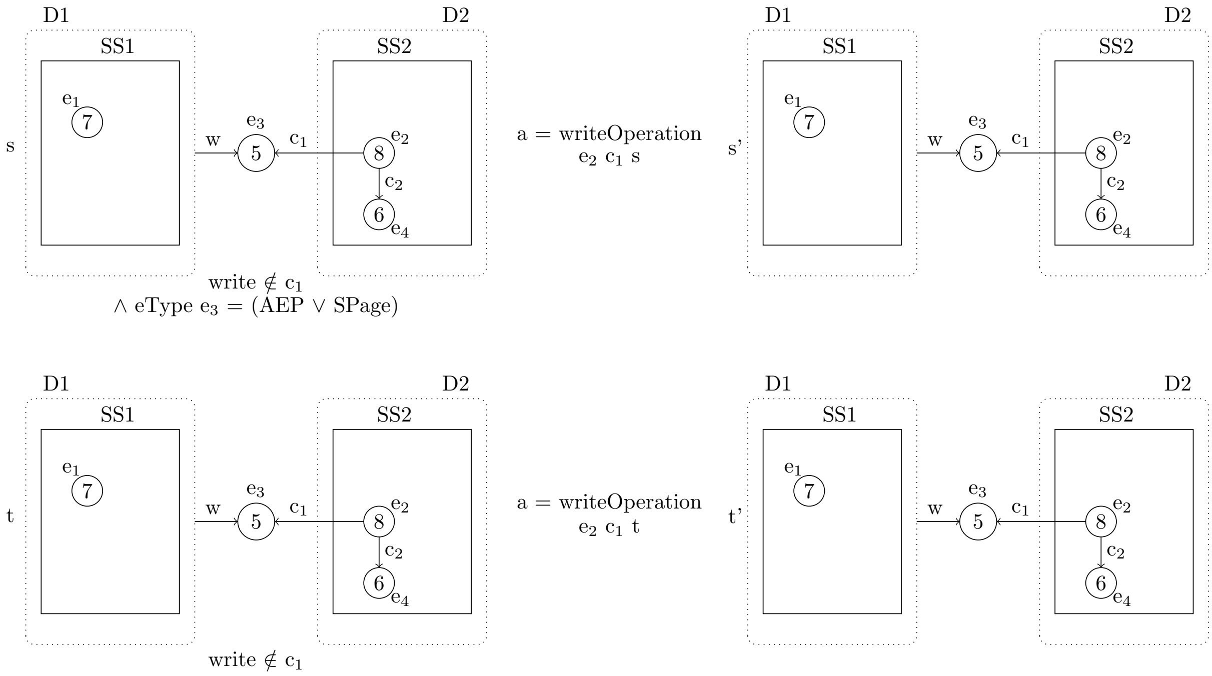

Figure 19: Noninterference for Write on an object = AEP executed from an entity ∈ Domain 2

*	$s \overset{D1}{\sim} t \Rightarrow$ aquiv_nonin s t D1
**	legal (SysWrite e_2 c_1) s = false $\Rightarrow$ s' = s
***	legal (SysWrite e_2 c_1) t = false $\Rightarrow$ t' = t

$\forall$ e $\in$ D1.

$$\text{value\_of s' e} \overset{**}{=} \text{value\_of s e} \overset{*}{=} \text{value\_of t e} \overset{***}{=} \text{value\_of t' e}$$
$$\wedge\ \text{caps\_of s' e} \overset{**}{=} \text{caps\_of s e} \overset{*}{=} \text{caps\_of t e} \overset{***}{=} \text{caps\_of t' e}$$
$$\wedge\ \text{subSys s' e} \overset{**}{=} \text{subSys s e} \overset{*}{=} \text{subSys t e} \overset{***}{=} \text{subSys t' e}$$

$\Rightarrow$ aquiv_nonin s' t' D1 $\Rightarrow s' \overset{D1}{\sim} t'$

7.3.4 Write on an AEP object from Domain 1

Write on AEP objects can be executed from Domain 1. Figure 20 showes that this has no influence on the noninterference property.

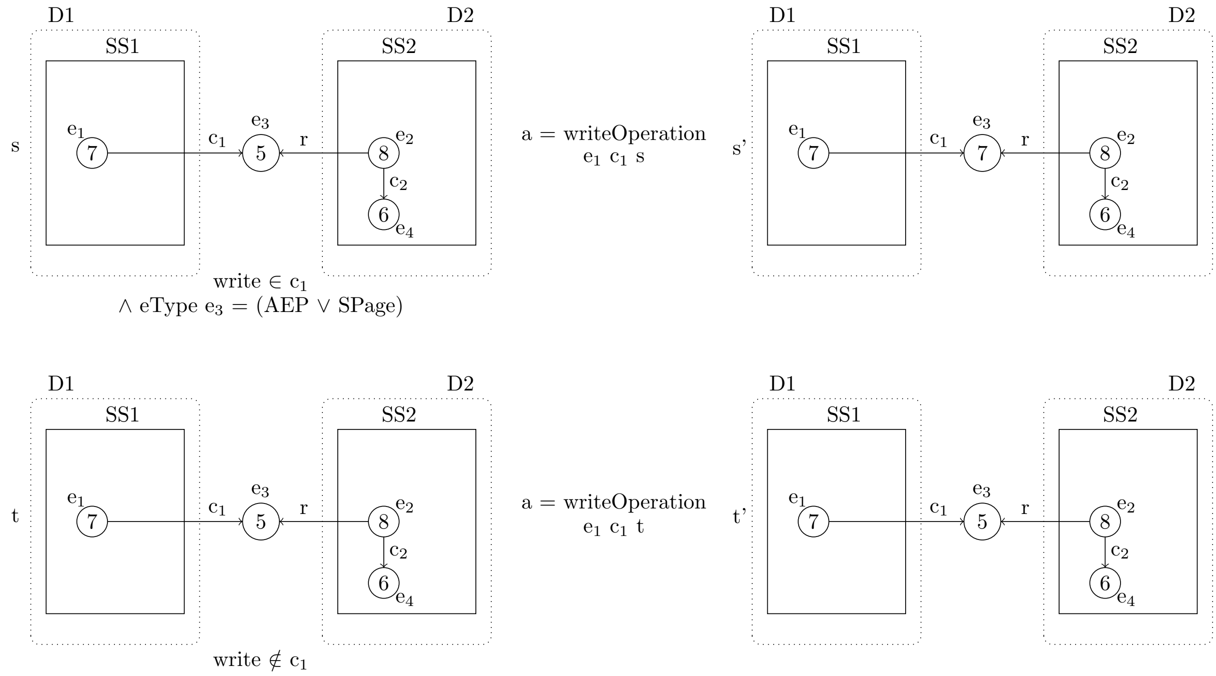

Figure 20: Noninterference for Write on an object = AEP executed from an entity ∈ D1

*	$s \overset{D1}{\sim} t \Rightarrow$ aquiv_nonin s t D1
**	writeOperation e_1 c_1 s changes the value $\in e_3 \notin$ D1.
	That means it has no impact on any entity ∈ D1
***	legal (SysWrite e_2 c_1) t = false ⇒ t' = t

$\forall$ e ∈ D1.

$$\text{value\_of s' e} \overset{**}{=} \text{value\_of s e} \overset{*}{=} \text{value\_of t e} \overset{***}{=} \text{value\_of t' e}$$

$$\wedge \quad \text{caps\_of s' e} \overset{**}{=} \text{caps\_of s e} \overset{*}{=} \text{caps\_of t e} \overset{***}{=} \text{caps\_of t' e}$$

$$\wedge \quad \text{subSys s' e} \overset{**}{=} \text{subSys s e} \overset{*}{=} \text{subSys t e} \overset{***}{=} \text{subSys t' e}$$

$\Rightarrow$ aquiv_nonin s' t' D1 $\Rightarrow s' \overset{D1}{\sim} t'$

7.4 Read

Read is legal on TCB, Sychronous IPC Endpoint and Asynchronous IPC Endpoint objects. Like in chapter 7.3 I distinguish between objects with legal execution of `read` on objects inside a domain, illegal execution of `read` on objects inside a domain an both on objects outside a domain.

7.4.1 Read on TCB or Sychronous IPC Endpoint objects

TCB and SEP objects are the two object types that are executable with `read` from an endpoint in the same domain.
Figure 21 shows how the operation influences the other domain.

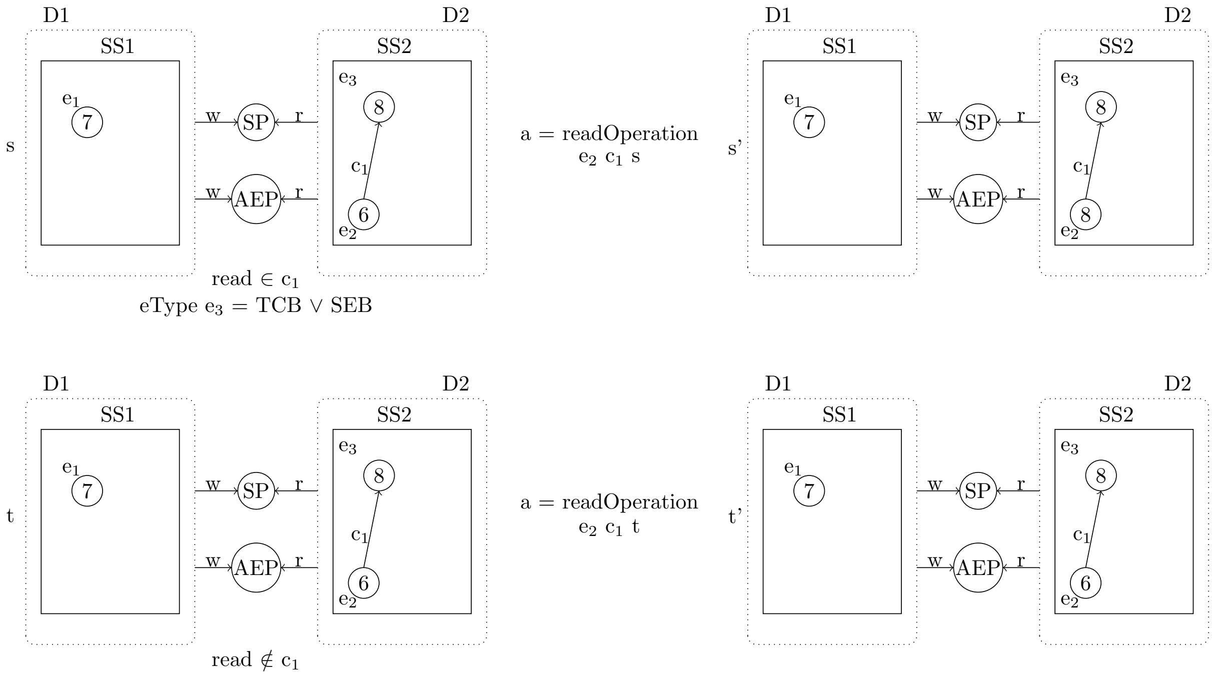

Figure 21: Noninterference for Read on a TCB or Sychronous IPC Endpoint object

*	$s \overset{D1}{\sim} t \Rightarrow$ aquiv_nonin s t D1
**	readOperation e_2 c_1 s only changes the value of an entity $\in$ D2 nothing in D1
***	legal (SysRead e_2 c_1) t = false $\Rightarrow$ t' = t

$$
\begin{aligned}
&\forall\ e \in D1. \\
&\quad \text{value\_of } s'\ e \overset{**}{=} \text{value\_of } s\ e \overset{*}{=} \text{value\_of } t\ e \overset{***}{=} \text{value\_of } t'\ e \\
&\wedge\ \text{caps\_of } s'\ e \overset{**}{=} \text{caps\_of } s\ e \overset{*}{=} \text{caps\_of } t\ e \overset{***}{=} \text{caps\_of } t'\ e \\
&\wedge\ \text{subSys } s'\ e \overset{**}{=} \text{subSys } s\ e \overset{*}{=} \text{subSys } t\ e \overset{***}{=} \text{subSys } t'\ e \\
&\Rightarrow \text{aquiv\_nonin } s'\ t'\ D1 \Rightarrow s' \overset{D1}{\sim} t'
\end{aligned}
$$

7.4.2 Read on other object types inside a domain

Figure 22 depicts the read operation on objects in the same domain on which `read` is not executable. It's similar to `write` in chapter 7.3.2.

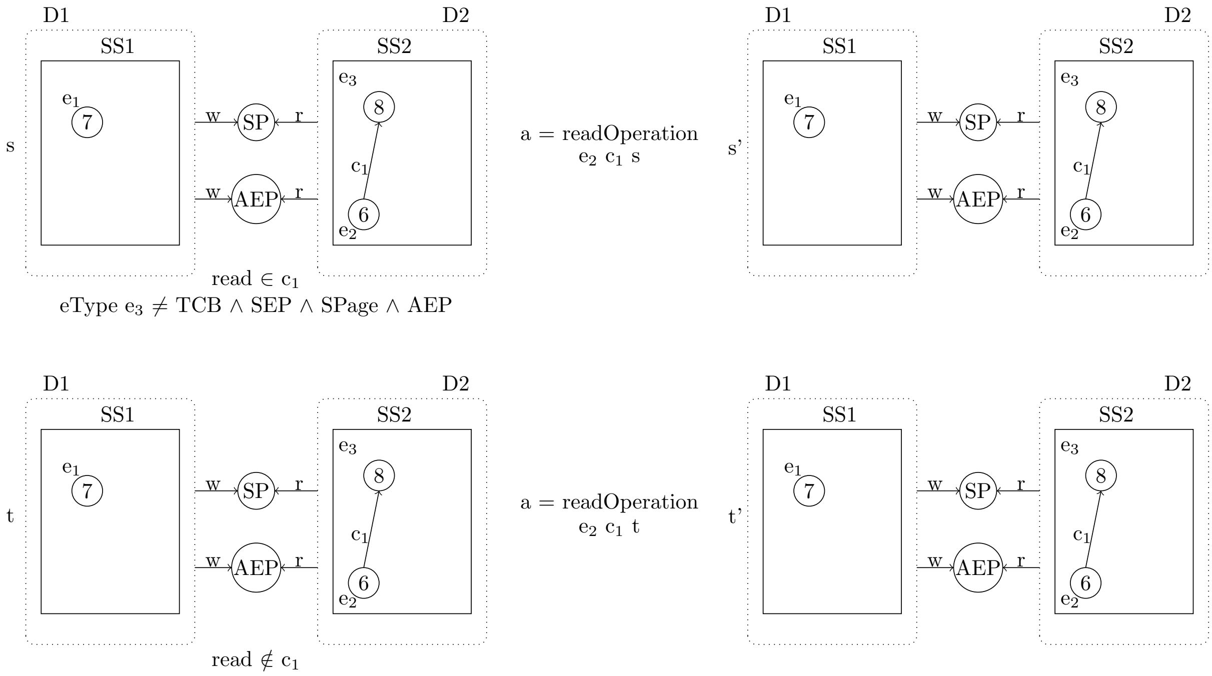

Figure 22: Noninterference for Read on object types $\neq$ TCB, Asynchronous IPC Endpoint or Sychronous IPC Endpoint object

*	$s \overset{D1}{\sim} t \Rightarrow$ aquiv_nonin s t D1
**	legal (SysRead e_2 c_1) s = false $\Rightarrow$ s' = s
***	legal (SysRead e_2 c_1) t = false $\Rightarrow$ t' = t

$\forall$ e $\in$ D1.

$$
\begin{aligned}
& \text{value\_of s' e} \overset{**}{=} \text{value\_of s e} \overset{*}{=} \text{value\_of t e} \overset{***}{=} \text{value\_of t' e} \\
\wedge\ & \text{caps\_of s' e} \overset{**}{=} \text{caps\_of s e} \overset{*}{=} \text{caps\_of t e} \overset{***}{=} \text{caps\_of t' e} \\
\wedge\ & \text{subSys s' e} \overset{**}{=} \text{subSys s e} \overset{*}{=} \text{subSys t e} \overset{***}{=} \text{subSys t' e}
\end{aligned}
$$

$\Rightarrow$ aquiv_nonin s' t' D1 $\Rightarrow$ $s' \overset{D1}{\sim} t'$

7.4.3 Read on an AEP object from Domain 1

Similar to chapter 7.3.3 `read` can only be executed from on type of Domain. Thats the one to which information is allowed to flow. In my case it's Domain 2. No infomation is allowed to flow to Domain 1. So `read` is not legal if it is executed from Domain 1.
Figure 23 shows that this does not affect Domain 1.

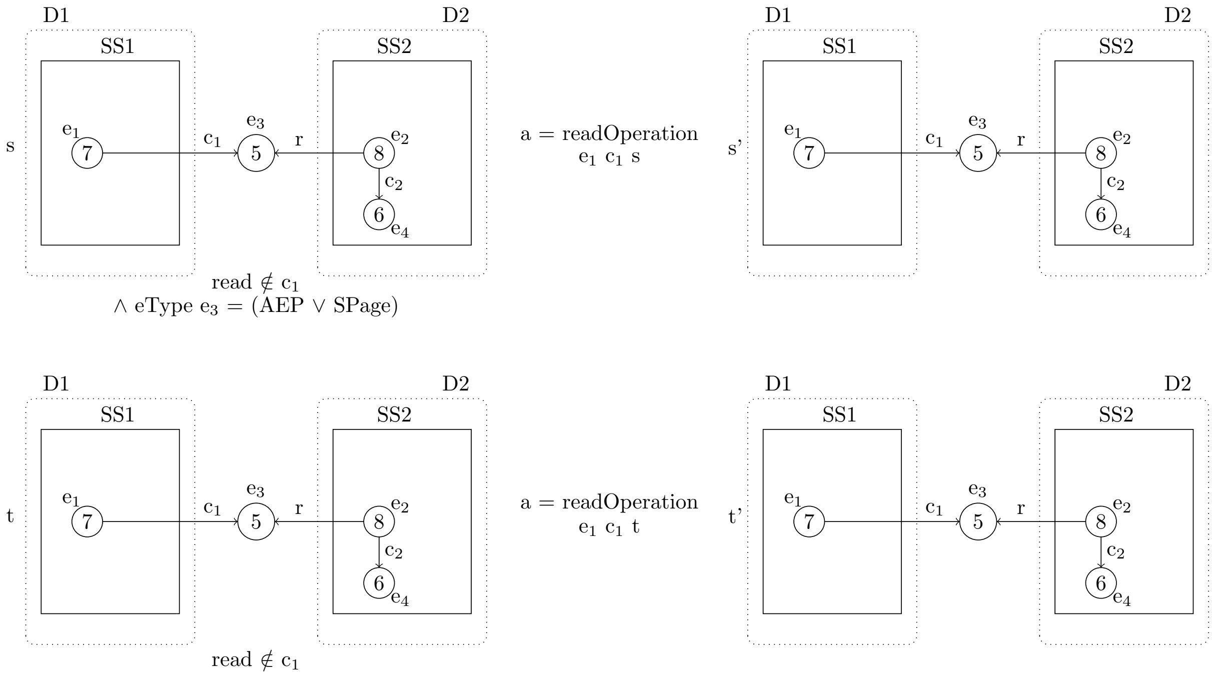

Figure 23: Noninterference for Read on object types = Asynchronous IPC Endpoint executed from Domain 1

*	$s \overset{D1}{\sim} t \Rightarrow$ aquiv_nonin s t D1
**	legal (SysRead e_1 c_1) s = false $\Rightarrow$ s' = s
***	legal (SysRead e_1 c_1) t = false $\Rightarrow$ t' = t

$\forall$ e $\in$ D1.

$$\text{value\_of s' e} \overset{**}{=} \text{value\_of s e} \overset{*}{=} \text{value\_of t e} \overset{***}{=} \text{value\_of t' e}$$

$$\wedge \quad \text{caps\_of s' e} \overset{**}{=} \text{caps\_of s e} \overset{*}{=} \text{caps\_of t e} \overset{***}{=} \text{caps\_of t' e}$$

$$\wedge \quad \text{subSys s' e} \overset{**}{=} \text{subSys s e} \overset{*}{=} \text{subSys t e} \overset{***}{=} \text{subSys t' e}$$

$\Rightarrow$ aquiv_nonin s' t' D1 $\Rightarrow s' \overset{D1}{\sim} t'$

7.4.4 Read on an AEP object from Domain 2

Read can be executed from Domain 2. In Figure 24 I show the impact of this execution.

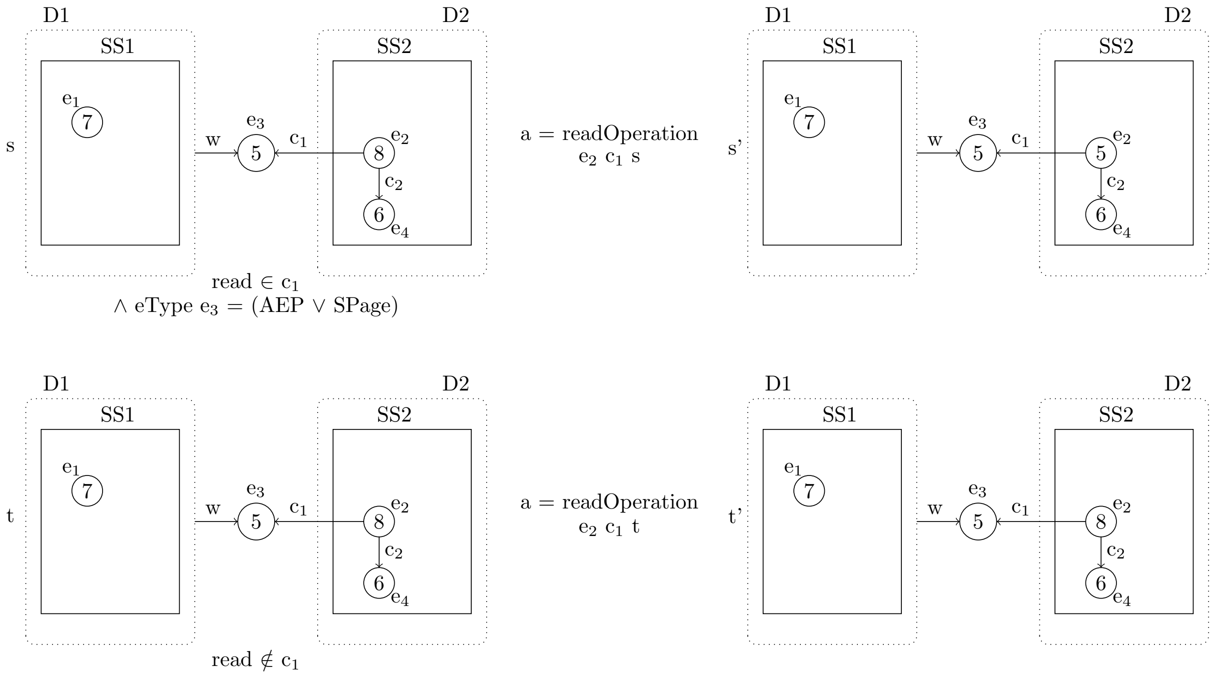

Figure 24: Noninterference for Read on object types = Asynchronous IPC Endpoint executed from Domain 2

*	$s \overset{D1}{\sim} t \Rightarrow$ aquiv_nonin s t D1
**	readOperation e_2 c_1 s changes the value $\in e_3 \notin$ D1. That means it has no impact on any entity $\in$ D1
***	legal (SysRead e_2 c_1) t = false $\Rightarrow$ t' = t

$\forall$ e $\in$ D1.

$$\text{value\_of s' e} \overset{**}{=} \text{value\_of s e} \overset{*}{=} \text{value\_of t e} \overset{***}{=} \text{value\_of t' e}$$
$$\wedge\ \text{caps\_of s' e} \overset{**}{=} \text{caps\_of s e} \overset{*}{=} \text{caps\_of t e} \overset{***}{=} \text{caps\_of t' e}$$
$$\wedge\ \text{subSys s' e} \overset{**}{=} \text{subSys s e} \overset{*}{=} \text{subSys t e} \overset{***}{=} \text{subSys t' e}$$
$$\Rightarrow \text{aquiv\_nonin s' t' D1} \Rightarrow s' \overset{D1}{\sim} t'$$

7.5 Remove

`Remove` can be executed on CNode, VSpace or Interrupt Controller object types.
Like in the chapters bevore I distinguish between executing the operation inside and outside a domain. All legal object types are inside a domain. So I only have to differ between legal and not legal for the execution inside da domain.

7.5.1 Remove on CNode, VSpace or Interrupt Controller objects

Remove deletes a capability in an entity. This capability can point on an entity in the same domain or on an AEP object.

- Target object is in the same domain If the removed capability points to an entity in the same domain and `remove` is legal for the executed entity, the operation looks like it is pictured in Figure 25.

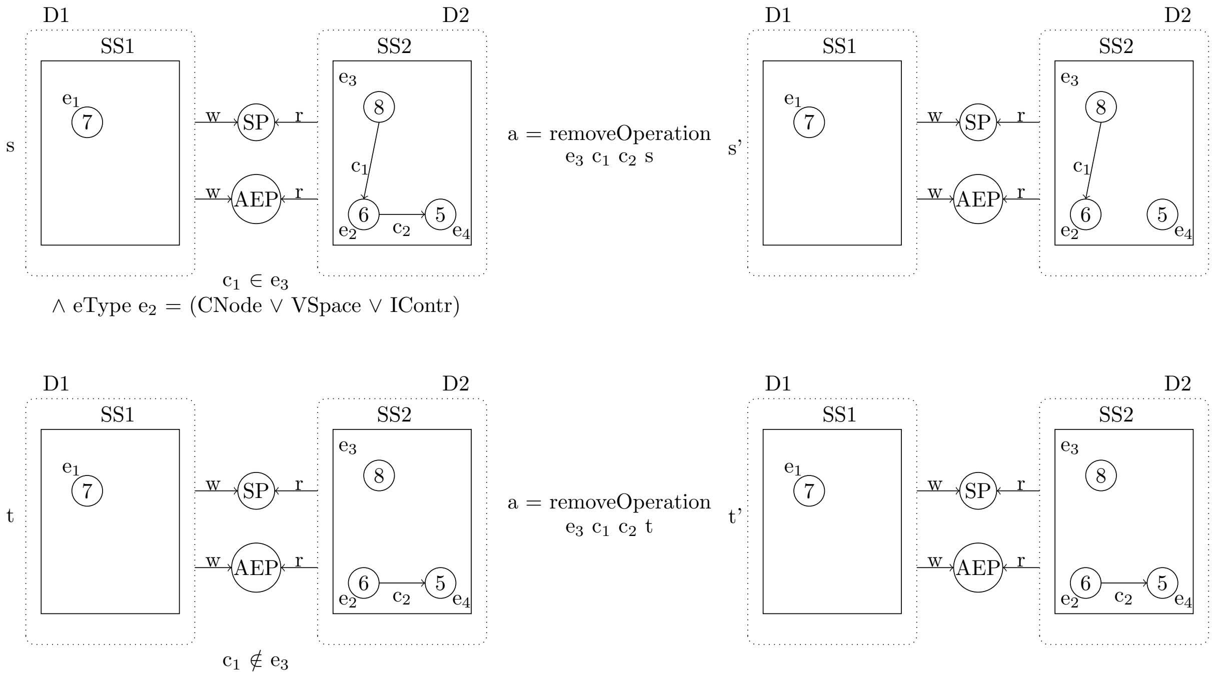

Figure 25: Noninterference for Remove on object types = CNode, VSpace or IContr. The removed capability points to an entity in the same domain

*	$s \overset{D1}{\sim} t \Rightarrow$ aquiv_nonin s t D1
**	readOperation e_2 c_1 s changes the value $\in e_3 \notin D1$. That means it has no impact on any entity $\in D1$
***	legal (SysRead e_2 c_1) t = false $\Rightarrow$ t' = t

$$
\begin{aligned}
&\forall\, e \in D1. \\
&\quad \text{value\_of } s'\ e \overset{**}{=} \text{value\_of } s\ e \overset{*}{=} \text{value\_of } t\ e \overset{***}{=} \text{value\_of } t'\ e \\
&\wedge\quad \text{caps\_of } s'\ e \overset{**}{=} \text{caps\_of } s\ e \overset{*}{=} \text{caps\_of } t\ e \overset{***}{=} \text{caps\_of } t'\ e \\
&\wedge\quad \text{subSys } s'\ e \overset{**}{=} \text{subSys } s\ e \overset{*}{=} \text{subSys } t\ e \overset{***}{=} \text{subSys } t'\ e \\
&\Rightarrow \text{aquiv\_nonin } s'\ t'\ D1 \Rightarrow s' \overset{D1}{\sim} t'
\end{aligned}
$$

7.6 Revoke

References

[1] T. Murray, D. Matichuk, M. Brassil, P. Gammie and G. Klein:
Noninterference for Operating System Kernels.
International Conference on Certified Programs and Proofs, pp. 126142, Kyoto, Japan, December, 2012

[2] D. Elkaduwe, G. Klein and K. Elphinstone:
Verified Protection Model of the seL4 Microkernel.
Technical Report NRL-1474, NICTA, October, 2007

[3] J. Andronick T. Bourke P. Derrin D. Greenaway D. Elkaduwe, G. Klein and K. Elphinstone R. Kolanski D. Matichuk T. Sewell S. Winwood:
Abstract Formal Specification of the seL4/ARMv6 API.
Version 1.3

[4] D. von Oheimb
Information flow control revisited: Noninfluence = Noninterference + Nonleakage.
In *9th ESORICS*, volume 3193 of *LNCS*, pages 225-243, 2004.

[5] D. Elkaduwe:
A Principled Approach To Kernel Memory Management.
PhD Thesis, UNSW CSE, Sydney, Australia, March, 2010

[6] M. Grosvenor and A. Walker:
seL4 Reference Manual.
Version 10.0.0

[7] G. Smith:
Principles of Secure Information Flow Analysis.
Chapter 13 (pp. 291-307) of Malware Detection, Springer-Verlag, 2007

[8] J.N. Buxton and B. Randell:
Software engeneering techniques.
Report on a conference sponsored by the NATO science committee, Rome, Italy, 27th to 31st October 1969