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, 07-27-2018

. . . .

Andrea Kuchar

Abstract

The thesis investigates the question whether the specification of the seL4 access control system is strong enough to verify Noninterference properties on it. I analyse the Take-Grant-Protection Model [1] and extend it for showing the Noninterference properties [6] on each of its system operations. As the specifications and proofs of the take-grant model are developed in the theorem proof assistant Isabelle/HOL, I use the same to formalise my datatypes and functions.

List of Figures

Contents

1 Introduction

1.1 Motivation

Nowadays our society becomes progressivly dependent on computer systems. Throughout our whole life smaller and smaller computers increasingly take over control. Wheter in a smart TV, our car or the lights in a connected home. We are therefore forced to confront ourselves with the safety and reliability of these systems.

This is particular essential when we entrust our lives to one of these computers. We expect on-board computers in cars or flight-computers to be free from defects and unhackable. Unfortunately the reality is often different. For example, hackers have proven that the onboard computer of some cars can be taken over from a smartphone in a nearby car.

A key component in developing secure systems is the operating-system (OS) kernel. The kernel has full access to hardware resources. One defect in the kernel can compromise the security and reliability of the entire system.

The weakness of most traditional kernels was their huge amount of code due to their monolithic design. This makes it hard to review or verify the code. Monolithic designs are fundamentally week because they integrate accessory functions like drivers for hardware or virtual filesystems. This makes the system more vulnerable for bugs. One crashed module can lead to a crash of the entire system.

In contrast, microkernels concentrate on the fundamental functions: interprocess communication, scheduling or memory management. The motivation behind microkernels is to reduce the possibility of bugs in the kernel code through reducing the code to an amount as minimal as possible and to exclude functions from kernel mode. With less code it becomes more feasible to guarantee the absence of defects within the kernel through formal verification.

Because we feed our smartphones, tablets, on-board computers, etc. with an ever growing amount of sensitive information like bank data, passwords, e-mails, chats the safety of embedded systems is a growing necessity.

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 detect an huge amount of bugs. But as Dijkstra said "Testing can only show the presence, not the absence, of bugs." [7]

As already mentioned, less lines of codes makes it more feasible to verify it relating to its specification. The seL4 microkernel is the first microkernel whose correctness is formally verified. It is a high-assurance, high-performance microkernel, primarily developed, maintained and formally verified by NICTA (now Trustworthy Systems Group at Data61) for secure embedded systems. Its 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 will explore if the extended take-grant model is strong enough to show noninterference properties 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 assures that there is no way information can flow to undesirable parties.
The thesis should investigate the different system operations of the model regarding the thereby occurring information flow.
With the collected information I want to answer two questions. First if the noninterference properties can be illustrated on the existing take-grant model and second if the noninterference properties are fulfilled or the different system operations the take-grant model provides.

1.3 Structure of the Thesis

At the beginning I want to give a survey of the seL4 kernel, its 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. Chapter 3 focuses on the formalisation of the take-grant model an Chapter 4 on the formalisation of the noninterference model.
From chapter 5 on I turn to the validation of the noninterference property. In chapter 7 the validation is subdivided into the different system operations. To show the property for the model I am going to extend the model in Chapter 6.
Finally I'll take a short resume and give a prospect on the possibilities to enhance this topic.

2 Requirements

2.1 The seL4 Microkernel

The seL4 [5] ist a small operation system kernel developed for the ARM11 arcitecture. All concepts, however, can be generalised to any architecture with a multilevel-page-table structure. [4] It is based on the L4 microkernel developed in the 1990s and proviedes a minimal number of services to applications, such as abstractions for virutal address spaces, threads, inter process comunication (IPC).

Each abstraction ist implemented by a kernel object with methods dependent on the abstraction it supplies. If an application whants to use one of the implemented services it to call the corresponding object through capabilities. They are stored in kernel objects called *CNodes*.

Each capability contains a 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 a corresponding method name, applications can invoke system calls. As arguments these system calls can have data or other capabilities. For example: If an object has a capability with write authority in it, pointing on a synchronous endpoint, it can send a message to another object, that has read authority on this endboint. It can do it by invoking the capability, with the write right in it, that points on the synchronous endpoint object. The other object in turn has to own and invoke a capability with the read right in it, that also point on the synchronous endpoint.

2.1.1 System Calls

The kernel provides the following system calls:

- `send()`: The system call argument is 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 is silently droped.

- `Call()`: Like `send()` but the application is blocked until the object provides a response, or the receiving application replies.

  If the 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 is 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 objects to allocate the system operations [5].

- **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 constructs 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 specified 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 is represented as a sibling of it.

  - `Move()`
    can move a capability between two specified slots.

  - `Mutate()`
    moves the capability similar to `Move()` and is able to reduce its rights as it is done in `Mint()` without an orignal copy remaining.

  - `Rotate()`
    moves two capabilities between three slots. Replaces 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, all children from the capability in the CDT are removed with `Delete()`.
  - `Recycle()`
    revokes all outstanding capabilities and reconfigures the 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 is always 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 thread control block object has the following methods:
  `CopyRegisters(), ReadRegisters(), WriteRegisters(), SetPriority(), SetIPCBuffer(), SetSpace(), Configure(), Suspend(), Resume()`

- **Virtual Memory**
  In the *virtual address space* (VSpace) Objects in the VSpace implement services for the management of virtual memory which largely directly correspond to those of the hardware:

  - seL4 ASID Table:
    The seL4 *Adress Space Identifier* table provides two services:

    1. With the seL4 ASID table address spaces where mappings have to be removed from can easily be detected, as the seL4 acts as an internal naming mechanism for them.
    2. In the seL4 PageDirectorys can be deleted without updating the capability links because the kernel sets the corresponding PageDirectory address in the ASID table to *NULL* if a PageDirectory is revoked. Therefore the seL4 ASID table provied so called *weak links* between capabilities on addresses in the ASID table and the VSpace mappings that are derived from them.

    The seL4 ASID table is a global, fixed-sized table that is created at boot time. Each seL4 ASID is associated with a hardware ASID.

- PageDirectory (PD):
  It defines the root page table of the two-level , hardware defined page table structure the ARM11 consists of. If an entity owns a authorised capability that points to a PD it has the right to manipulate the related VSpace.

- PageTable (PT):
  The leaf node of the ARM11, two-level page table structure is implemented by the PageTable object. A page table entry contains either an invalid entry, or a pointer to a 4 kilobyte Page.

- Pages or Frames:
  A virtual memory page is implemented by region of physical memory called a *Page* or *Frame*. The name differs from paper to paper. In the specification it is called a *Page* [2]. It has three methods:

  - If it gets an PD or PT capability as an argument the `map` installs a PD-entry or PT-entry and refers to the Page in the specified location.

  - If the permissions of an existing mapping have to be changed the Page calls the `remap` method.

  - The `unmap` method is used to remove an existing mapping.

The following illustrates the creation of a VSpace:

1. First a PD object has to be allocated with the retype operation on untyped memory objects.

2. The *Resource Manager* has to initialise it with a seL4 ASID table:

   - It invokes the PD object

   - It passes a capability that allows the use of a slot in the seL4 ASID table. The memory address of the PD is copied into the provided ASID table slot.

   - The capability pinting on the PD is updated by storing the sel4 ASID table index into it instead of the PD index.

3. Now the PD can be used as a VSpace of one or more threads.

4. The ressource manager can install a PageTable into an adress space by invoking a PageDirectory and passing a PageTable capability and the virtual address. After installing a PageTable the kernle stores the seL4 AID of the corresponding PD and the virtual address in the PT capability.

5. With Pages the kernel proceed in a similar way.

Figure 2.1 shows how the objects of a VSpace are connected.

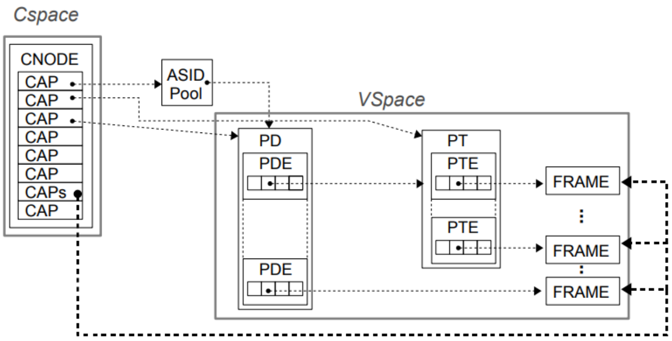

Figure 2.1: Internal representation of an application in seL4 [2]

- **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)` encapsulate 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 are created. It also returns capabilities to the new objects if it succeeds.

2.1.3 Memory Allocation Model

A special characteristic of the seL4 is that the memory for kernel objects is not allocated dynamically. A goal was to isolate physical memory access between applications and to control the amount of physical memory that applications can use.
To accomplish this applications get fixed sized memory regions they have to control by themselves.
Capabilities on Untyped Memory Objects (UMO) are needed to create new objects. So applications need the capabilities on UMOs to create new objects. After creation the objects have a fixed amount of memory they can use.
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 kernel then creates an *Initial User Thread* with associated CSpace and VSpace and hands over the remainig memory in form of capabilities on UMOs.
The Initial User Thread can create smaller sized UMOs out of an UMO or `retype` it into another object type. The creator of new objects has full authority over the objects. This "full authority" depends on the object type.
Figure 2.2 shows a sample system architecture in which a resource manager running at user-level has the authority over the remaining untyped memory after bootstrapping.

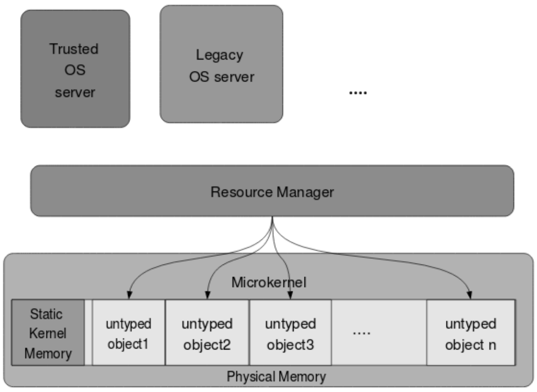

Figure 2.2: Sample system configuration [1]

2.2 The Take-Grant Model

Protection or acces control models specify, analyse and implemente security policies. The classical Take-Grant Model was primary introduced by Lipton and Snyder, 1977 in "A Linear Time Algorithm for Deciding Subject Security" [8].

2.2.1 The Classical Model

In the Take-Grant Model [1] subjects or objects are represented as nodes and authority as arcs in a directed graph that represents the system.
Rules for graph mutation represent the different system operations to modify the authority distibution. 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 $\in \gamma$. "t" denotes the take authority.

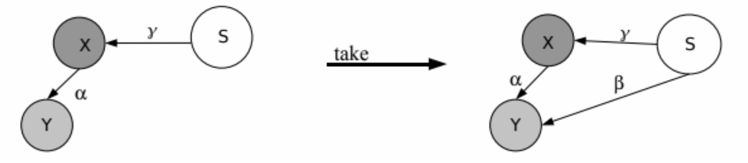

Figure 2.3: *Take* adds an edge from S to Y with the label $\beta \subseteq \alpha$. [1]

- **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 $\in \gamma$. "g" denotes the grant authority.

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

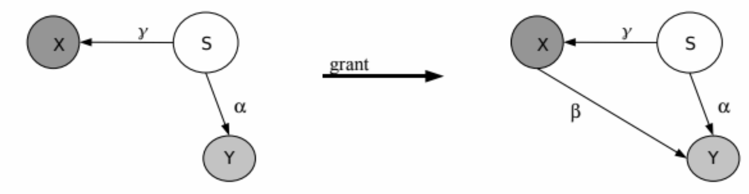

Figure 2.4: *Grant* adds an edge from X to Y with the label $\beta \subseteq \alpha$. [1]

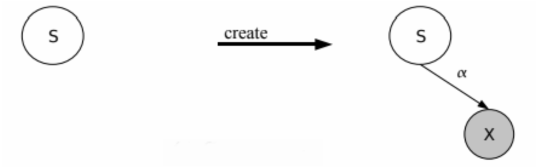

Figure 2.5: *Create* adds a new node X and an arc from S to X, labelled with α. [1]

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

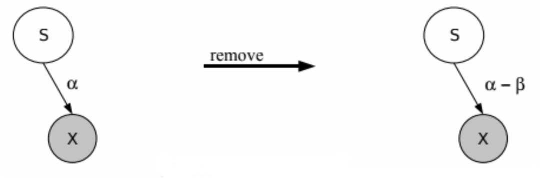

Figure 2.6: *Remove* deletes β labels from α or the arc itself if $\alpha - \beta = \{\}$. [1]

2.2.2 Take-Grant specified for the seL4

The Take-Grant Model specified in the paper "Verified Protection Model of the seL4 Microkernel" [1] is a variant of the classical Take-Grant model. In the paper the developers extended the original model in several ways.

From the modifications made the one on the *create rule* is the most important one. As I explained in chapter 2.1 authority in the kernel is implemented with capabilities. Adding a new node to the protection graph in the model corresponds to the creation of a new object with a capability pointing on it in the kernel. Therefore the object executing the `create operation` needs a capability with `create` authority.

The *remove rule* was modified as it does not remove parts of labels anymore but the whole capability. That means the complete arc pointing on an object is removed.

To diminish authority a capability has to be removed and newly created with diminished authority.

With `retype` newly created capabilities are saved in a *Capability Derivation Tree* (CDT) as children of the UMO. A capability can be copied with the `mint` or `imitate operation`.

A capability copied with `mint` is inserted in the CDT as child of the original one. Those that are copied with `imitate` are siblings. Figure 2.7 showes a CDT where C1 and C2 are created from the UMO via `retype`. C3 and C4 are copied from C1 via `mint`. So they have the same or less authority as C1. C1' is copied from C1 via `imitate`. This operation transfers the same rights to the new capability. As a consequence the capability is inserted a sibling of C1.

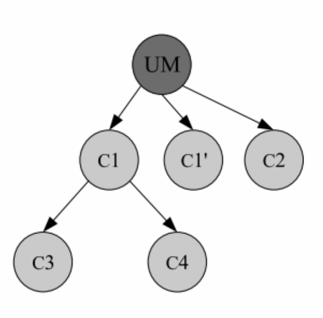

Figure 2.7: Example CDT with children and siblings [4]

To remove a set of capabilities the operation `revoke` was implemented.
With this operation `remove` is executed on every capability that is in the CDT below the target capability.
The take rule was removed from the model and the grant rule was not modified:
An entity e_1, that has a capability with the grant right in it, pointing to an entity e_2 can give the same or less amount of rights on another entity to e_2.
A speciality of the extended model is, that objects and subjects are called *entities*.
The goal of the paper "Verified Protection Model of the seL4 Microkernel" was to show that implementing isolated subsystems, using the mechanisms of the seL4 kernel, can be accomplished. [1]
Isolated subsystems are implemented as a collection of *connected* entities. An entity that has *grant authority* on another one is connected with this entity. Authority can neither get in nor get out of these isolated subsystems.
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. It ensures that objects and subject from different security levels do not interfere with those at other levels. In the model variables are classified to be L (low security) or H (high security, private) variables. The goal is to prevent information to flow from H variables to L variables.
I use the noninterference formulation as it is used in Geoffrey Smiths Principles of Secure Information Flow Analysis. [6], which reads "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 that, if in a program two states are equivalent on a low level domain,

then they are still equivalent on this level after a program was executed.
As they are equivalent, also if the program was not executed in one of the states, this implies that the low level domain not only is not able to get the information of the program but even can not recognize if the program was executed. The execution of the program in a high level domain has no impact on the low level domain. This implies that no information flows from the high level to the low level domain.
Central to noninterference is the notion of a *policy* $\rightsquigarrow$. It specifies the allowed information flows between domain. $L \rightsquigarrow H$ if information is allowed to flow from domain L to domain H. The model says two memories μ and ν agree on L variables if they fullfil an equivalence relation $\mu \overset{\mathrm{L}}{\sim} \nu$.
The exact formalisation of noninterference for the validation follows in Chapter 4.

3 Formalisation of the Take-Grant Model

3.1 Capabilities

In the Take-Grant model for seL4 [1], where I got the formalisation from, the authors waived the usual differentation between 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 are 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 that names a target-entity
2. A set of rights that defines which system operations the source-entity is authorized 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 fields:

1. The `heap`, which stores the entities of the system like an array from address `0` up to and excluding `next_id`.

2. `next_id` contains slot for the next entity without overlapping with an existing one.

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

3.2 System Operations

The data type `sysOps` defines the different system operations of the seL4.

```
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 is the passed capability and for `SysRemove` it is the one that has to be removed. The rights set in `SysGrant` is 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"
```

The function `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 `step'` and `step` functions define the execution of a single system operation. The original executions of `SysNoOp`, `SysRead` and `SysWrite` do not have an underlying function. All other functions are defined.

The `step` function:

```
step' ::  "sysOPs ⇒ state ⇒ state" where
     "step'   (SysNoOp e) s            =   s"
  |  "step'   (SysRead e c) s          =   s"
  |  "step'   (SysWrite e c) s         =   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 defined functions for the system operations `create, grant, remove` and `revoke`:

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


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⦈)⦈"
```

4 Formalisation of the Noninterference Model

For the validation I need a formalisation of the noninterference property.

noninterference :: "bool" where
"noninterference ≡ ∀ *a l h s t s' t'*.reachable *s* ∧ reachable *t* ∧ *s* $\overset{l}{\sim}$ *t* ∧ (*h* ⇝ *l*
⟶ *s* $\overset{h}{\sim}$ *t*) ∧ (*s,s'*)∈ Step *a* ∧ (*t,t'*)∈ Step *a* ⟶ *s'* $\overset{l}{\sim}$ *t'*”

”a” names the system operation, ”l” a low level domain, ”h” a high level domain, from the states ”s” and ”t” the system operation is executed and ”s'” and ”t'” are the resulting states.

The definition says that if the states s and t of a system are eyuivalent for a low level domain they still have to be equivalent for this domain after the execution of a program a, if the following preconditions are fulfilled:

- s and t are reachable in the system. That means there exists a command sequence as such that the state s is reachable with as from the initial state s_0 and a command sequence bs such that the state t is reachable with bs from s_0.
- If the policy says that information is allowed to flow from h to l, s and t have to be equivalent for the high level domain h.
- The program a has to be defined for the states s and t.

First I tried to validate confidentiality for the different system operations as they are defined in the take-grant-model. With this model it is impossible to decide whether a change of value has been recognized by another domain.
In the paper an entity only includes a set of capabilites. For my purpose I need the option to access the content of the entities, 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 extend the original record entity by adding a *value* modelled by a natural number.

My entity type:

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

I also had to modify the formalisation of the system operations as they should be able to `read` and `write` the value of an entity. For that they need a fuction that returns the value of an entity:

```
value_of ::  "state ⇒ entity_id ⇒ nat" where
"value_of s sref ≡ eValue(heap s sref)"
```

The modified `read`, `write`, `create`, `grant` and `remove` operations:

```
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|))|)"

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)⦈)⦈"
```

The modified `step` and `step'` functions:

```
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
```

To check noninterference I had to define a few functions.

1. The isolation by using subsystems:
   For the definition of Subsystems I need the option to check whether the execution of a sequence of commands can lead to a state where an entity can leak information to the other. To ensure that we controll information flow over the whole system , only the symmetric, refelexive, transitive closure of the `leak` function is useful.
   The `leak` fuction [1] indicates that in a state `s` , an entity `e` has the ability to give a capability to an entity e_i. Capabilities can be handed over with the `SysGrant` or `SysCreate` operations. They are legal if the initiating entity has at least `grant` rights in the capability, pointing to the modified entity. So the `leak` function checks if the entity `e` has at least `grant` authority on entity $\mathtt{e}_i$.
   The symmetric closure is implemented by the fuction `connected`, written $\mathtt{s} \vdash \mathtt{e} \longleftrightarrow \mathtt{e}_i$. It is true, if there exists a leak from `e` to $\mathtt{e}_i$ or from $\mathtt{e}_i$ to `e`.
   The tarnsitive closure is indicated with $\mathtt{s} \vdash \mathtt{e} \longleftrightarrow\!* \ \mathtt{e}_i$.
   `in_conc_connected s e` $\mathtt{e}_i$ is true if the tansitive closure of `connected` for `e` and $\mathtt{e}_i$ in state s is true.
   Now I can define subsystems. First I need a new datatype `subSusT` to identify subsystems:

   ```
   type_synonym  subSysT = "entity_id set"
   ```

   The following function defines subsystems:

   ```
   subSys ::  "state ⇒ entity_id ⇒ subSysT" where
   "subSys s e ≡ {∀ e_i.in_conc_connected s e e_i}"
   ```

2. The equivalence relation "∼":
   $\mathtt{s} \overset{\mathtt{d}}{\sim} \mathtt{t}$ means that for every entity `e` reachable from an entity in the subsystem `d` the status of `e` in `s` and `t` has to be the same.
   An entity `e` is in a subsystem `d`, expanded by an entity $\mathtt{e}_i$ in state `s`, if $\mathtt{e} \in$ `subSys s` $\mathtt{e}_i$.

   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_of s e = value_of t e ∧ caps_of s e =
   caps_of t e ∧ subSys s e = subSys t e}"
   ```

5 Validation of Noninterference

After the formalisations in Chapter 3 and 4 I try to establish noninterference for the different system operations.

First I tried to identify the conditions under which noninterference holds. For this I constructed a small model of the system with one Low-level-Subsystem and one High-level-Subsystem, with entities in them and tested for different right-sets and different operations if the noninterference-property holds.
The following displays this approach for the `write` operation from a low level domain on a high level domain.
I assume:

- H denotes a High level domain that implements the subsystem 'H'
- L denotes a Low level domain that implements the subsystem 'L'
- e_1 is an entity in H and e_2 is an entity in L
- $s \overset{L}{\sim} t$

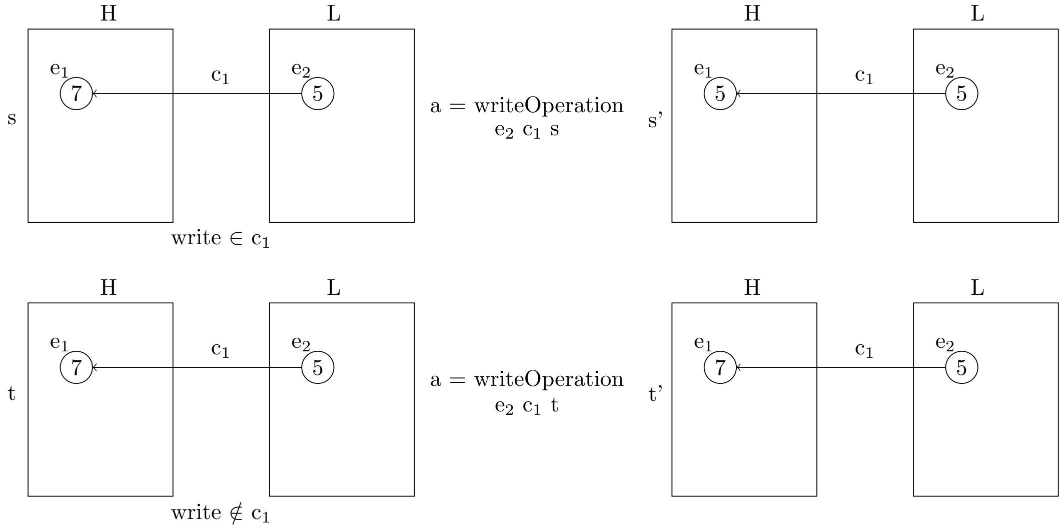

Figure 5.1: Noninterfernce of Write 1

To check noninterference I examine if the criterias for `aquiv_nonin s' t' L` are fulfilled after the execution of the `write operation` and the named preconditions.
The `write operation` in the extended model satisfies the noninterference property:

- We have `value_of s' e = value_of s e` $\wedge$ `caps_of s' e = caps_of s e` and `subSys s' e = subSys s e` as the `write operation` on the entity, $\mathtt{c}_1 \in e_2$ is pointing on, changes an entity $\mathrm{e}_1 \in$ H and does not affect an entity $\in$ L.

- We have `value_of s e = value_of t e` $\wedge$ `caps_of s e = caps_of t e` and `subSys s e = subSys t e` as one of the preconditions was s $\overset{L}{\sim}$ t. I defined the equivalence relation with the function aquiv_nonin s t L, which is equal to the requirement.

- The `step` function first checks whether the execution of the system operation is legal, if not the new state t' equals the old state t.
  `value_of t e = value_of t' e` $\wedge$ `caps_of t e = caps_of t' e` and `subSys t e = subSys t' e` as `write` is not part of c_1. So legal(SysWrite e_2 c_1) s = false what leads to t=t'.

In the following cases the proof looks always the same. So I shorten it:

Preconditions:

*	s $\overset{L}{\sim}$ t $\Rightarrow$ aquiv_nonin s t L
**	writeOperation e_2 c_1 changes $\mathrm{e}_1 \in$ H no e $\in$ L
***	legal(SysWrite e_2 c_1) t = false $\Rightarrow$ t=t'

Proof of the noninterference property for Write 1:

For all e$\in$L, we have:

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

Hence aquiv_nonin s' t' L , what is equivalent to s' $\overset{L}{\sim}$ t'

With s' $\overset{L}{\sim}$ t' the noninterference property for `write` is fulfilled.

6 Redesign of the Take-Grant-Model

This procedure worked until I came to the remove-operation. There I noticed the issue, 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:

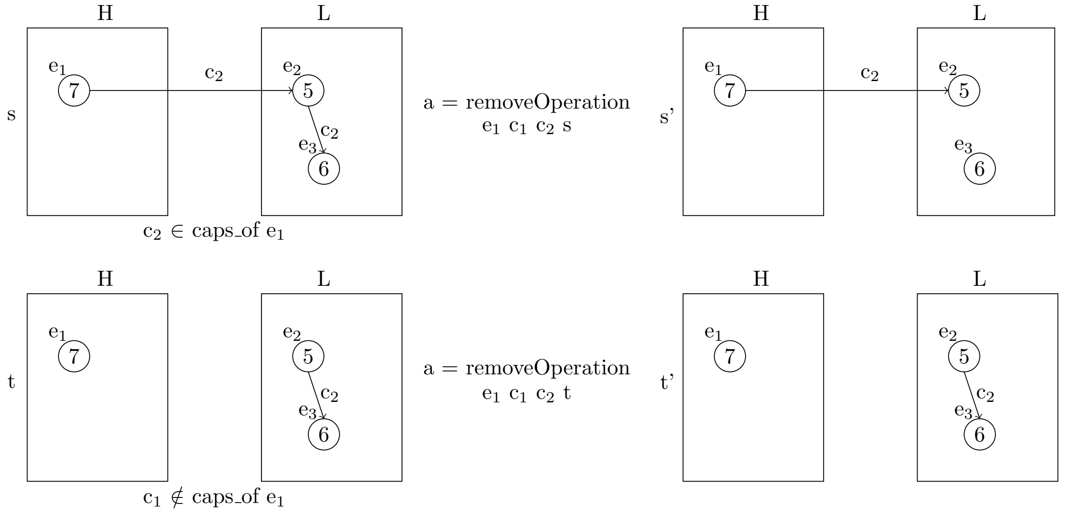

Figure 6.1: No confidentiality for Remove

In this example the L domain knows that the `removeOperation` was performed in the H domain as the capability c_2 was deleted. As a consequence the noninterference property is not achieved.

To study that problem I decided to classify the entities by their types, corresponding to the kernel specification [5]:

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

- Shared Pages (Pages (Frames) can be shared between domains. The corresponding capability has to be copied and then mapped in the VSpace of the other domain.)

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

Capability Type	Concrete Kernel	Take-Grant-Model
Untyped	Retype Revoke	sequence of *SysCreate* *SysRevoke*
TCB	ThreadControl 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*
InterruptController	Register interrupt Unregister interrupt	*SysGrant* *SysRemove*
Interrupt Handler	Acknowledge interrupt	*SysWrite*

Table 6.1: Relationship: operation of concrete kernel ⟷ of protection model [4]

To discern 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
                   |   SPage
                   |   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 ∨ SPage"
 |  "legal   (SysWrite e c) s        =   (isEntityOf s e ∧ c ∈ caps_of s e ∧ Write
                                         ∈ rights c) ∧ eType (entity c) = TCB ∨ SEP ∨
                                         AEP ∨ IHandl ∨ SPage"
 |  "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 whether 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* and *Shared Pages*.
Figure 6.2 pictures an example of how the objects and methods can be placed in the domains and how the connection to *Asynchronous Endpoints* and *Shared Pages* is implemented if the information is allowed to flow from domain 1 to domain 2: D1$\rightsquigarrow$D2 but D2$\not\rightsquigarrow$D1.
The connections between the objects inside a domain are just examples. Exept from those from each TCB to a VSpace and a CSpace. At boot time the initial resource-manager gives authorities to the applications. If information is allowed to flow from domain 1 to domain 2, domain 1 gets write rights on the shared SPage and AEP object and domain 2 only read on both of them.

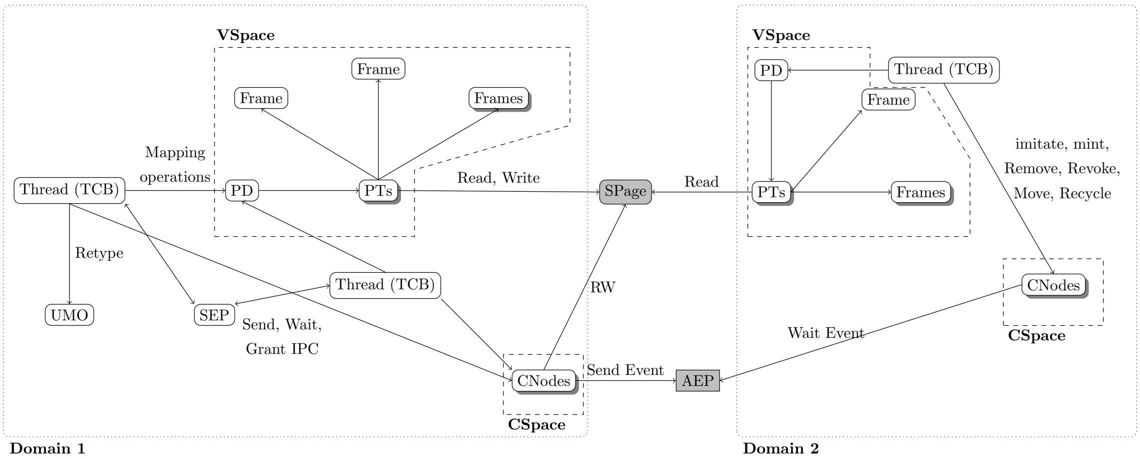

Figure 6.2: Objects and methods in the kernel

7 Validation with the new model

Now I whant to show that noninterference is valid for all system operations in the new defined model.

Theorem:
Let `s` an `t` be well defined states. That means:

- *They are reachable from the initial state s_0.*
- *They only include the in Chapter 6 defined object types.*
- *The capabilities pointing on `Shared Pages (SPage)` and `Asynchronous IPC Endpoints (AEP)` correspond to the rights the `Initial Resource Manager (IRM)` assigned the domains at boottime, corresponding to the given information flow policy. I this case D1$\rightsquigarrow$D2.*
- *If information is also allowed to flow from domain 2 to domain 1 then s and t are equivalent for domain 2.*

If `s` and `t` are equivalent for domain 1 (s $\overset{D1}{\sim}$ t) the the states `s'` and `t'` are still equivalent for D1 (s' $\overset{D1}{\sim}$ t'), after the execution of a system operation `a`.

Lemma:
If the information flow policy allows information to flow from D1 to D2 the IRM gives D1 write right on the `SPage` and `AEP` and D2 read rights on both of them. If it is allowed to flow from D2 to D1 it is the other way arround.
As the only way to get new rights on an object is through the `SysCreate` or `SysGrant` operations and the only connection out of daomains are those to the `SPages` and `AEPs`, the maximum of rights adomain can get on these objects is the set of rights it got from the `IRM` at boot time.

Proof: I examine each operation of the protection model and distinguish therefore between the different object types.
For this I assume that Domain 1 equates a low level domain and Domain 2 a high level domain. So information is allowed to flow from Domain 1 to Domain 2 but not from Domain 2 to Domain 1:`D1`$\rightsquigarrow$`D2 but D2`$\not\rightsquigarrow$`D1`

Further I assume:

- State s is equivalent to state t for Domain 1, what is represented by the function $\texttt{aquiv\_nonin:}\ \ \texttt{s} \overset{\texttt{D1}}{\sim} \texttt{t} \equiv \texttt{aquiv\_nonin s t D1}$
- The states s and t are reachable from the initial state s_0.
- The included object types correspond to those defined in Chapter 6.
- Because of *Lemma 1* I can assume that the connections on `SPages` and `AEPs` are well defined as claimed in *Theorem 1*.

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

7.1 Create

Create corresponds to the *Retype* operation on untyped memory objects (UMOs). Each Domain has a own and fixed section of memory. So the UMO for the `retype` is located in the same Domain as the implementing entity. Furthermore 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 a `H` domain changes or not the equivalence criteria in the `L` domain that is not allowed to get information from the former one.

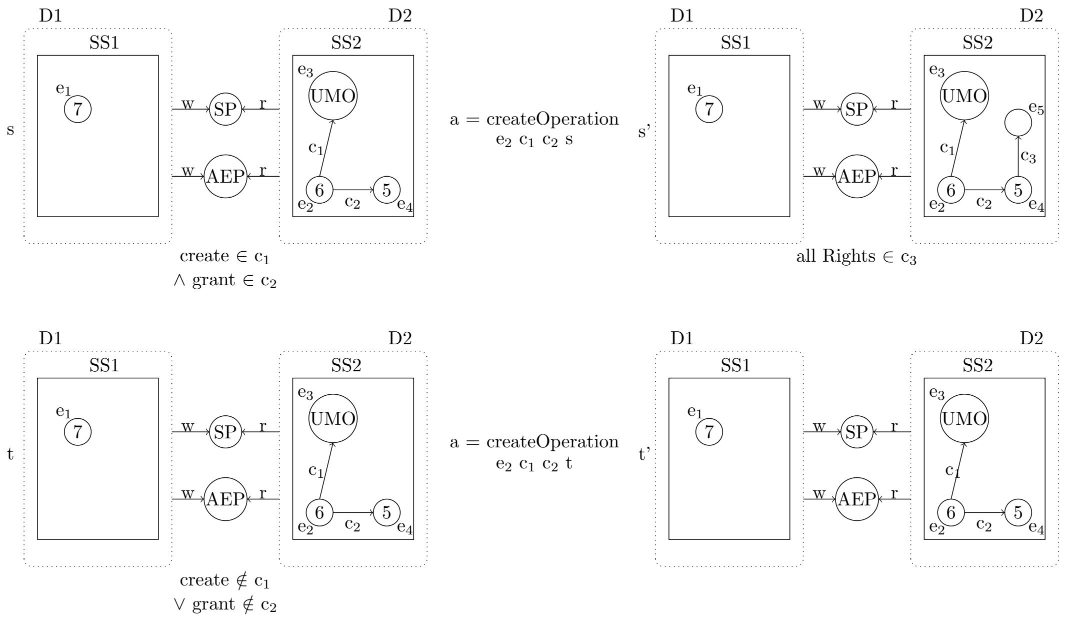

Figure 7.1: Noninterference for Create on Untyped Memory Objects

Each domain includes a subsystem. D1 includes the subsystem SS1 and D2 the subsystem SS2. Each figure has four subfigures. The first one in the left upper corner showes the state s before the execution of the `create` operation. The one on the right

of it shows state s' after the execution of the `create` operation with the changed subsystems. The subfigure in the left lower corner showes the state t before the execution of the `create` operation. The one on the right of it shows state t' after the execution of the `create` operation with the changed subsystems.
I have to show that if s $\overset{D1}{\sim}$ t and (s,s') ∈ Step a and (t,t') ∈ Step a then s' $\overset{D1}{\sim}$ t'. s $\overset{D1}{\sim}$ t was defined in Chapter 7 as the boolean fuction aquiv_nonin s t D1. The function is true if all entities e ∈ 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 check if 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 ∈ D1 after the execution of createOperation e_2 c_1 c_2 s respectively createOperation e_2 c_1 c_2 t. If that is the case I can say that aquiv_nonin s' t' D1 = true. From my definition of `aquiv_nonin` this leads to s' $\overset{D1}{\sim}$ t'.

Preconditions:

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

Proof of the noninterference property for create on UMO:

For all e ∈ D1, we have:

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

Hence aquiv_nonin s' t' D1 , what is equivalent to 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. So the new state `s'` equates the old state `s`.
The following figure shows the createOperation for every other object type inside a domain.

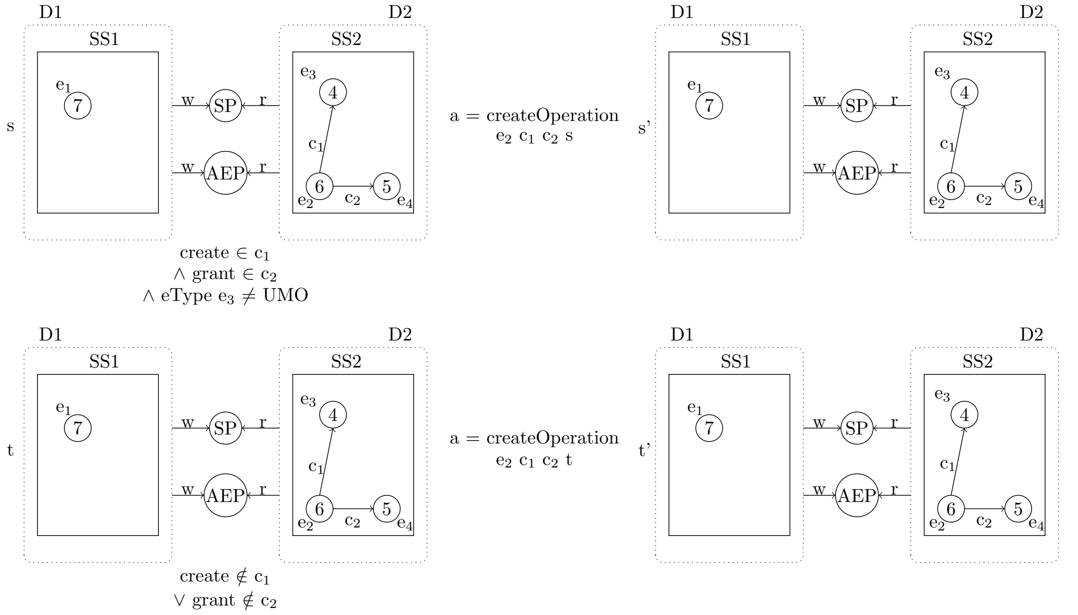

Figure 7.2: Noninterference for Create on object types ≠ Untyped Memory Objects

Preconditions:

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

Proof of the noninterference property for create on other object types in a domain:

For all e ∈ D1, we have:

(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)

Hence aquiv_nonin s' t' D1 , what is equivalent to $s' \overset{D1}{\sim} t'$

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 or Shared Page objects

Next I want to be sure that create has no impact on the entities in the L domain if it is executed on AEP or SPage objects.

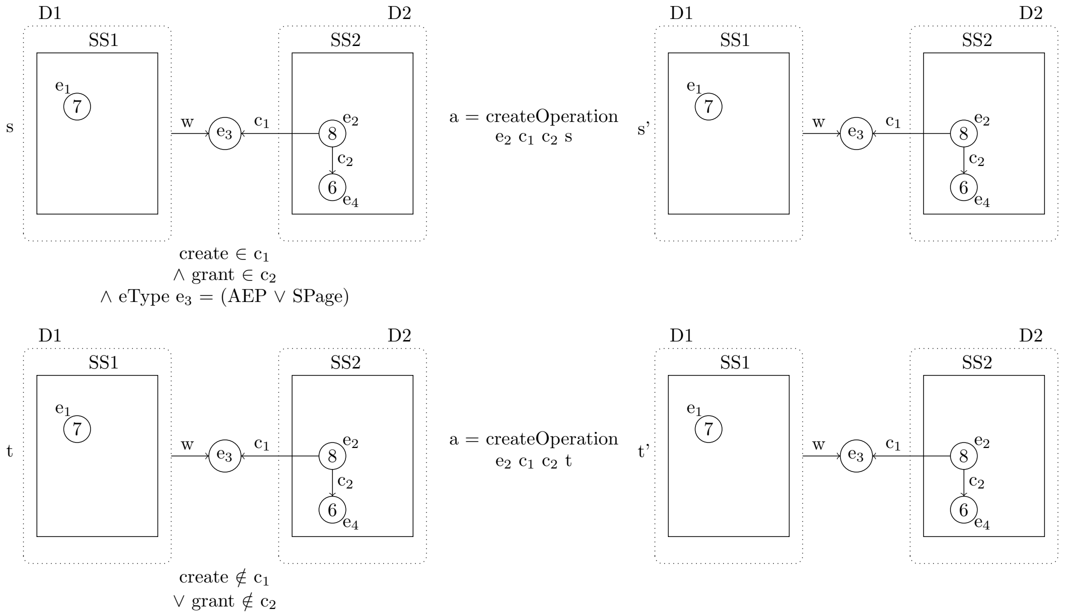

Figure 7.3: Noninterference for Create on object types = AEP ∨ SPage

In this case the check if the execution is legal = false in both states. So in both states the step' function leads to the definition of the old state.

Preconditions:

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

Proof of the noninterference property for `create` on Asynchronous IPC Endpoint or Shared Page objects:

For all e ∈ D1, we have:

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

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

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

Hence aquiv_nonin s' t' D1 , what is equivalent to $s' \overset{D1}{\sim} t'$

With $s' \overset{D1}{\sim} t'$ the noninterference property for `create` on Asynchronous IPC Endpoint or Shared Page objects is fulfilled.

7.2 Grant

The `grant` operation can only be performed inside a domain on a TCB, Synchronous IPC, CNode, VSpace or Interrupt Controller object.

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

Now I show that any grant operation inside a H domain on one of the named objects does not affect the values, capabilities or entities of an L domain.
Because every given object behaves in the same way, I generalized e_4 = TCB $\vee$ SIPC $\vee$ CNode $\vee$ VSpace $\vee$ IContr.

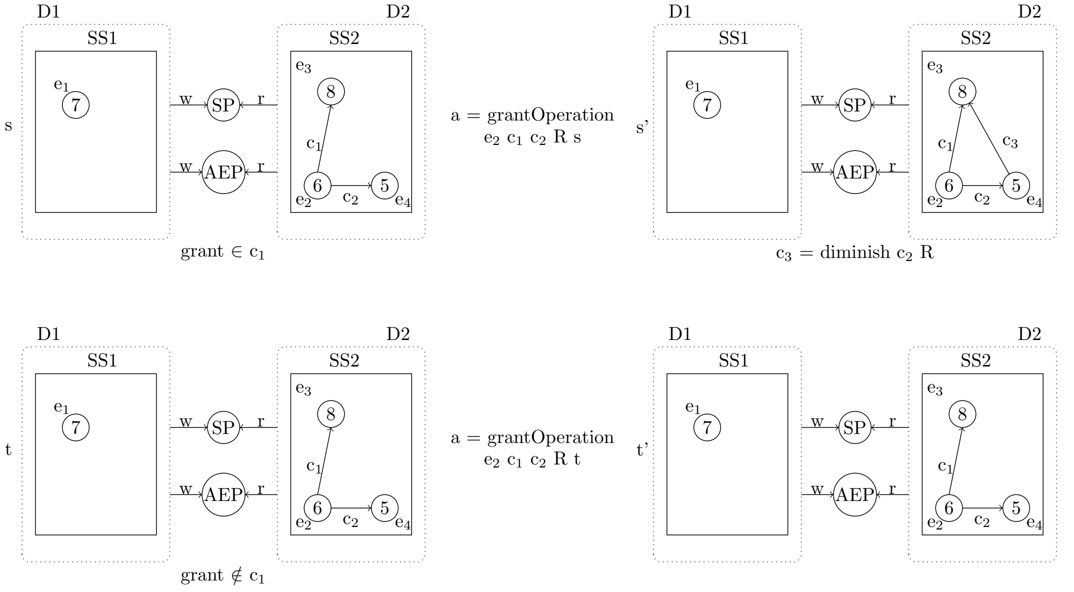

Figure 7.4: Noninterference for Grant on an TCB, Synchronous IPC Endpoint, CNode, VSpace or Interrupt Controller object

The the `grant` operation has no impact on the entities of the other domain because it adds a capability to an entity inside the same domain as the target and source entities are.

Preconditions:

* $s \overset{D1}{\sim} t \equiv$ aquiv_nonin s t D1

** grantOperation e_2 c_1 c_2 R s creates $c_3 \in$ D2 and does not change or create any capability $\in$ D1

*** legal (SysGrant e_2 c_1 c_2 R) t = false $\Rightarrow$ t' = t

Proof of the noninterference property for `grant` on TCB, Synchronous IPC Endpoint, CNode, VSpace and Interrupt Controller objects:

For all e $\in$ D1, we have:

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

Hence aquiv_nonin s' t' D1 , what is equivalent to s' $\overset{D1}{\sim}$ t'

With s' $\overset{D1}{\sim}$ t' the noninterference property for `grant` on TCB, Synchronous IPC Endpoint, CNode, VSpace and Interrupt Controller objects is fulfilled.

7.2.2 Grant on other objects inside a domain

In this paragraph I check if an execution of the grant operation on another object than TCB, SEP, CNode, VSpace, Interrupt Controller or the object types that establish a communication interface between domains: AEP and SPage, alters the configuration of the other domain.

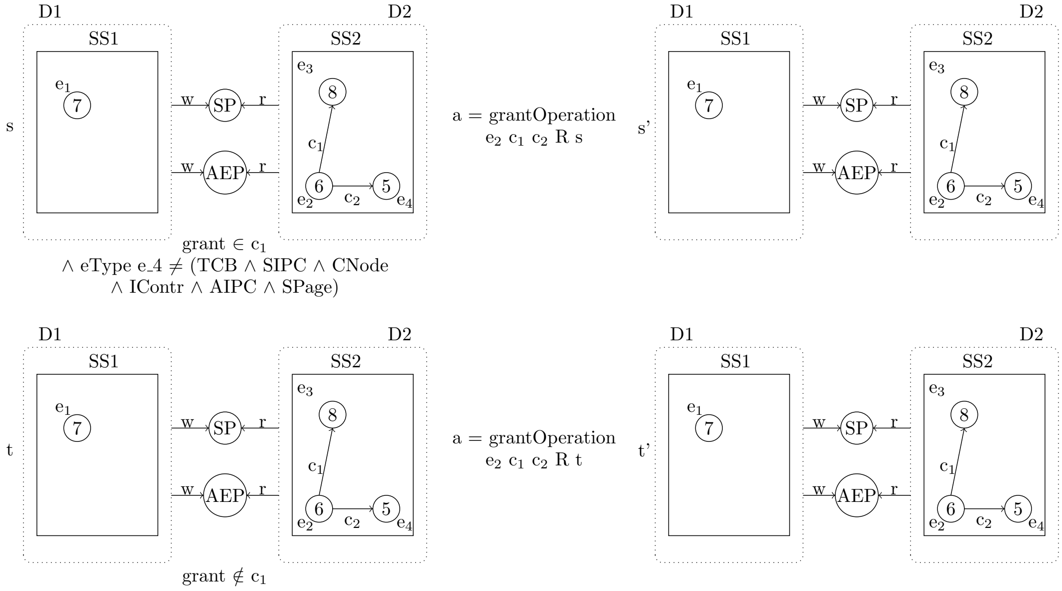

Figure 7.5: Noninterference for Grant on an object ≠ TCB, SEP, CNode, VSpace, IContr, SPage or AEP object

Preconditions:

*	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 R) t = false ⇒ t' = t

Proof of the noninterference property for `grant` on an object ≠ TCB, SEP, CNode, VSpace, IContr, SPage or AEP:

For all e ∈ D1, we have:

(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)

Hence aquiv_nonin s' t' D1 , what is equivalent to s' $\overset{D1}{\sim}$ t'

With s' $\overset{D1}{\sim}$ t' the noninterference property for `grant` on an object ≠ TCB, SEP, CNode, VSpace, IContr, SPage or AEP is fulfilled.

7.2.3 Grant on Asynchronous IPC Endpoint or Shared Page objects

The next figure illustrates grant on the two object types connecting different domains. In both cases the operation is not legal. So the new state equates the old one.

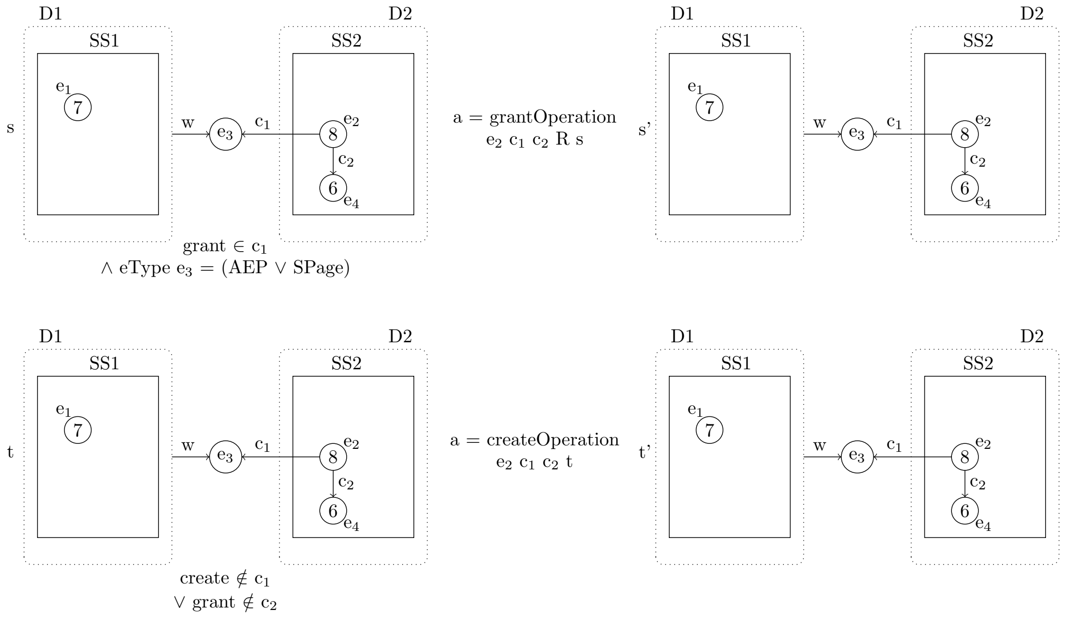

Figure 7.6: Noninterference for Grant on an Asychronous IPC Endpoint object

Preconditions:

*	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

Proof of the noninterference property for `grant` on an object = SPage or AEP:

For all e ∈ D1, we have:

(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)

Hence aquiv_nonin s' t' D1 , what is equivalent to s' $\overset{\text{D1}}{\sim}$ t'

With s' $\overset{\text{D1}}{\sim}$ t' the noninterference property for grant on an object = SPage or AEP is fulfilled.

7.3 Write

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

7.3.1 Write on TCB, SEP or IHandl objects

I start with the `write` operation on all executable objects inside a domain. So in the next figure e_3 = TCB ∨ SEP ∨ IHandl.

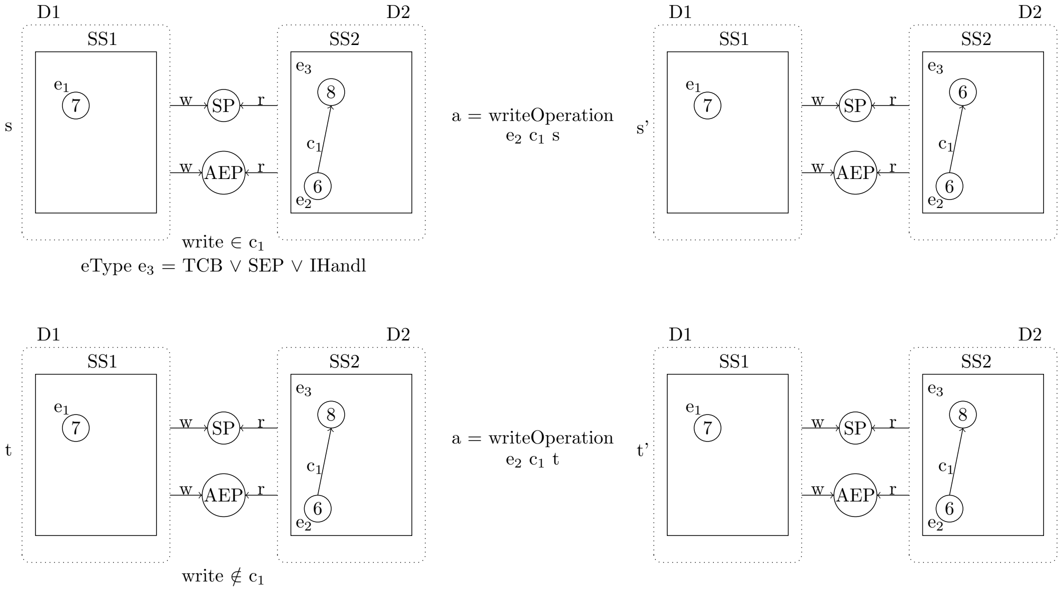

Figure 7.7: Noninterference for Write on a TCB, Sychronous IPC Endpoint or Interrupt Handler object

Preconditions:

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

Proof of the noninterference property for `write` on TCB, SEP or IHandl objects:

For all e ∈ D1, we have:

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

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

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

Hence aquiv_nonin s' t' D1 , what is equivalent to $s' \overset{D1}{\sim} t'$

With $s' \overset{D1}{\sim} t'$ the noninterference property for `write` on TCB, SEP or IHandl objects is fulfilled.

7.3.2 Write on objects $\neq$ TCB, SEP, IHandl, SPage 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 Controllers.
`Write` operation on these objects:

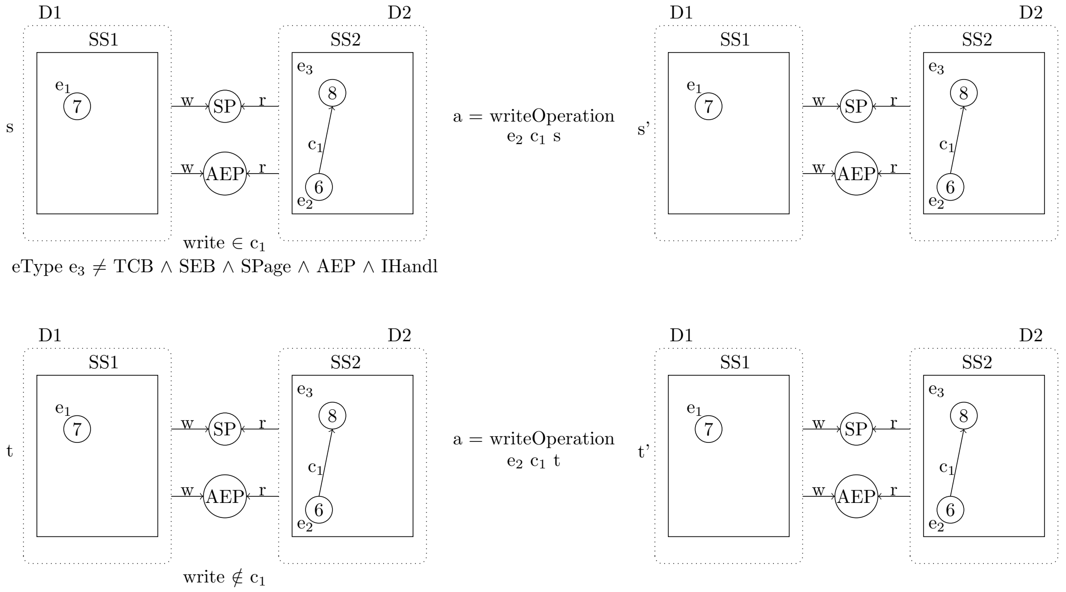
D1
D2
SS1
SS2
s
e_1
7
w
SP
r
AEP
e_3
8
c_1
6
e_2
a = writeOperation e_2 c_1 s
s'
write $\in$ c_1
eType e_3 $\neq$ TCB $\wedge$ SEB $\wedge$ SPage $\wedge$ AEP $\wedge$ IHandl
t
a = writeOperation e_2 c_1 t
t'
write $\notin$ c_1

Figure 7.8: Noninterference for Write on objects $\neq$ TCB, SEP, IHandl, SPage and AEP

Preconditions:

*	$s \overset{D1}{\sim} t \equiv$ 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

Proof of the noninterference property for `write` on objects $\neq$ TCB, SEP, IHandl, SPage and AEP:

For all e $\in$ D1, we have:

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

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

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

Hence aquiv_nonin s' t' D1 , what is equivalent to $s' \overset{D1}{\sim} t'$

With $s' \overset{D1}{\sim} t'$ the noninterference property for `write` on objects $\neq$ TCB, SEP, IHandl, SPage and AEP is fulfilled.

7.3.3 Write on AEP or SPage objects from Domain 2

In Chapter **??** I defined the precondition ⇒ $D1 \rightsquigarrow D2$ but $D2 \not\rightsquigarrow D1$. That means the rights from Domain 2 on Asychronous Endpoints and Shared Pages are restricted to `read`. If the write operation is called from Domain 2 it looks like it is illustrated in Figure 7.9. The policy prescribes that information is only allowed to flow from Domain 1 to Domain 2 but not from Domain 2 to Domain 1. As a consequence `write` can not be part of c_1.

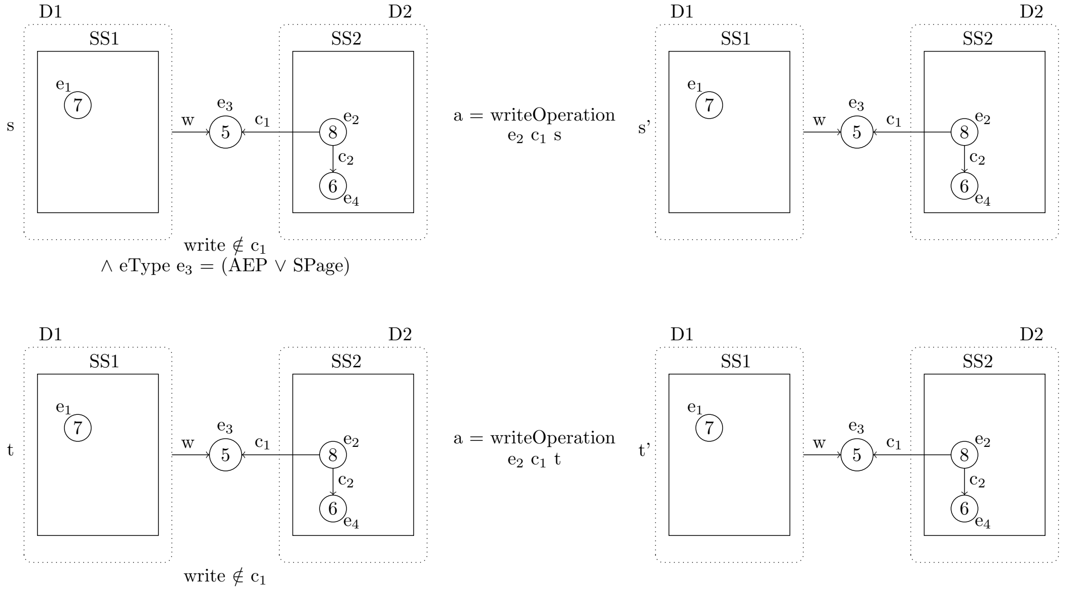

Figure 7.9: Noninterference for Write on an object = AEP executed from an entity ∈ Domain 2

Preconditions:

* $s \overset{D1}{\sim} t \equiv$ aquiv_nonin s t D1

** legal (SysWrite e_2 c_1) s = false ⇒ s' = s

*** legal (SysWrite e_2 c_1) t = false ⇒ t' = t

Proof of the noninterference property for `write` on AEP or SPage objects from Domain 2:

For all e ∈ D1, we have:

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

Hence aquiv_nonin s' t' D1 , what is equivalent to $s' \overset{D1}{\sim} t'$

With $s' \overset{D1}{\sim} t'$ the noninterference property for `write` on AEP or SPage objects from Domain 2 is fulfilled.

7.3.4 Write on an AEP or SPage object from Domain 1

Write on AEP objects can be executed from Domain 1. Figure 7.10 shows that this has no influence on the noninterference property.

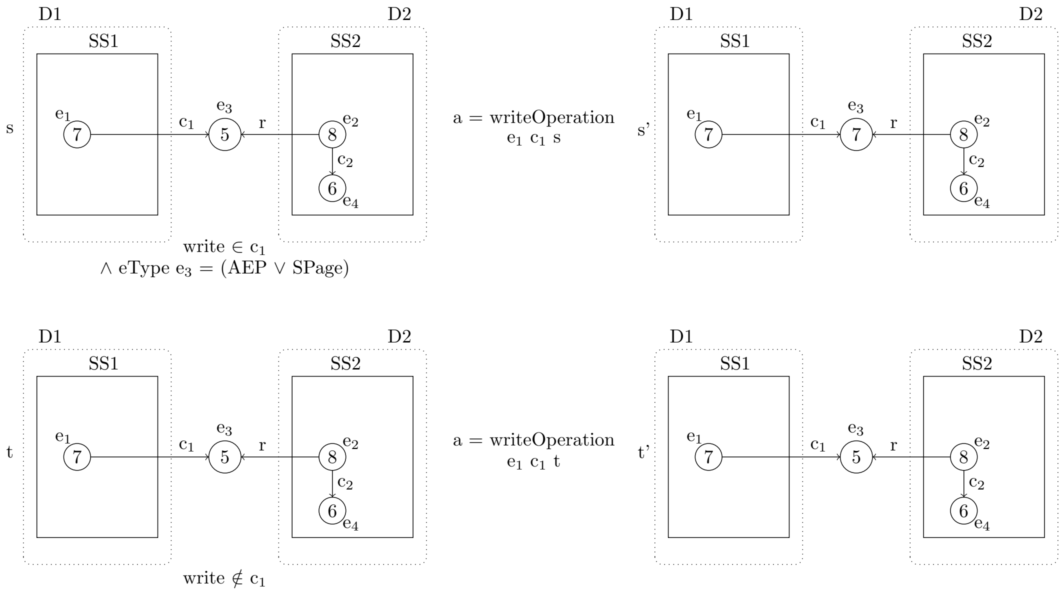

Figure 7.10: Noninterference for Write on an object = AEP executed from an entity ∈ D1

Preconditions:

*	$s \overset{D1}{\sim} t \equiv$ 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

Proof of the noninterference property for `write` on AEP or SPage objects from Domain 1:

For all e ∈ D1, we have:

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

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

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

Hence aquiv_nonin s' t' D1 , what is equivalent to $s' \overset{D1}{\sim} t'$

With $s' \overset{D1}{\sim} t'$ the noninterference property for `write` on AEP or SPage objects from Domain 1 is fulfilled.

7.4 Read

Read is legal on TCB, Sychronous IPC Endpoint, Asynchronous IPC Endpoint and Shared Page 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 and both on objects outside a domain.

7.4.0.1 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 7.11 illustrates whether the operation influences the L domain.

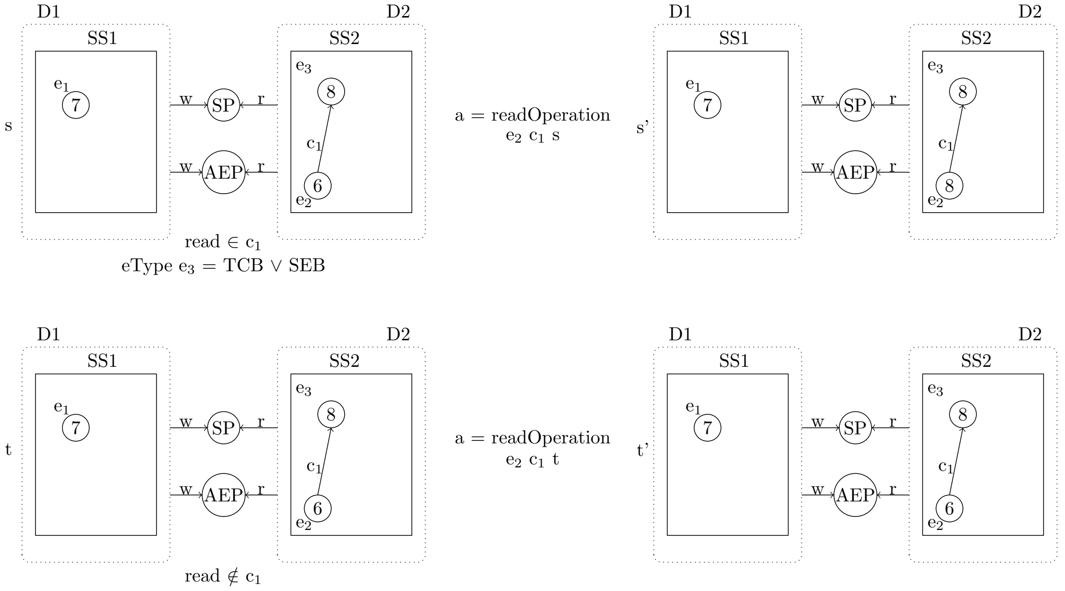

Figure 7.11: Noninterference for Read on a TCB or Sychronous IPC Endpoint object

Preconditions:

*	$s \overset{D1}{\sim} t \equiv$ 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

Proof of the noninterference property for `read` on TCB or SEP objects:

For all e $\in$ D1, we have:

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

Hence aquiv_nonin s' t' D1 , what is equivalent to $s' \overset{D1}{\sim} t'$

With s' $\overset{D1}{\sim}$ t' the noninterference property for `read` on TCB or SEP objects is fulfilled.

7.4.1 Read on other object types inside a domain

Figure 7.12 depicts the read operation on objects in the same domain on which `read` is not executable. It is similar to `write` in chapter 7.3.2.

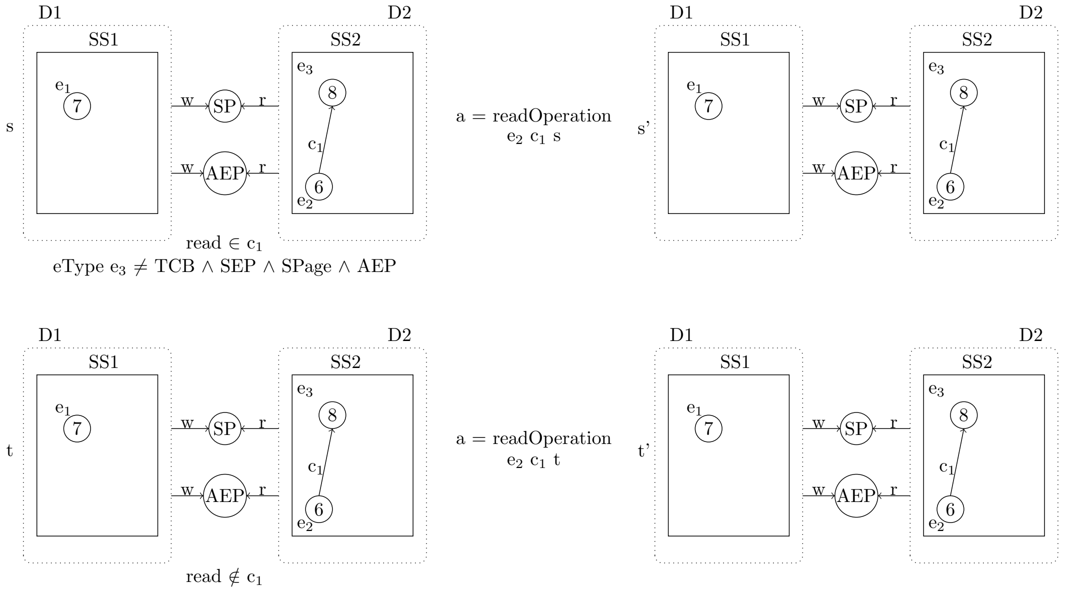
D1
D2
SS1
SS2
s
e_1
7
e_3
8
c_1
6
e_2
w
SP
r
w
AEP
r
read ∈ c_1
eType e_3 ≠ TCB ∧ SEP ∧ SPage ∧ AEP
a = readOperation e_2 c_1 s
s'
a = readOperation e_2 c_1 t
t
t'
read ∉ c_1

Figure 7.12: Noninterference for Read on objects ≠ TCB, Asynchronous IPC Endpoint, Sychronous IPC Endpoint or Shared Page

Preconditions:

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

Proof of the noninterference property for `read` on objects ≠ TCB, SEP, SPage and AEP:

For all e ∈ D1, we have:

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

Hence aquiv_nonin s' t' D1 , what is equivalent to s' $\overset{D1}{\sim}$ t'

With s' $\overset{D1}{\sim}$ t' the noninterference property for `read` on objects ≠ TCB, SEP, SPage and AEP is fulfilled.

7.4.2 Read on AEP or SPage objects from Domain 1

Similar to chapter 7.3.3 `read` can only be executed from a `H` domain. That is the one to which information is allowed to flow. In my case it is domain 2. No information is allowed to flow to domain 1. So `read` is not legal if it is executed from domain 1. Figure 7.13 shows that this does not affect domain 1.

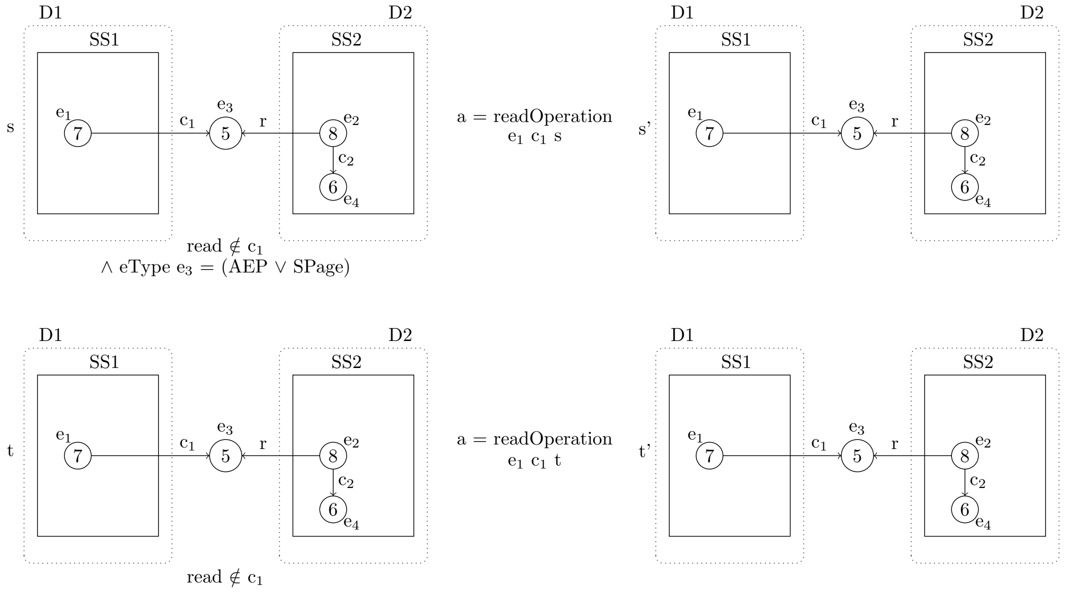

Figure 7.13: Noninterference for Read on object types = Asynchronous IPC Endpoint executed from Domain 1

Preconditions:

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

Proof of the noninterference property for `read` on AEP or SPage objects from Domain 1:

For all e ∈ D1, we have:

(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)

Hence aquiv_nonin s' t' D1 , what is equivalent to s' $\overset{D1}{\sim}$ t'

With s' $\overset{D1}{\sim}$ t' the noninterference property for `write` on AEP or SPage objects from Domain 1 is fulfilled.

7.4.3 Read on AEP or SPage objects from Domain 2

Read can be executed from Domain 2. In Figure 7.14 I show the impact of this execution.

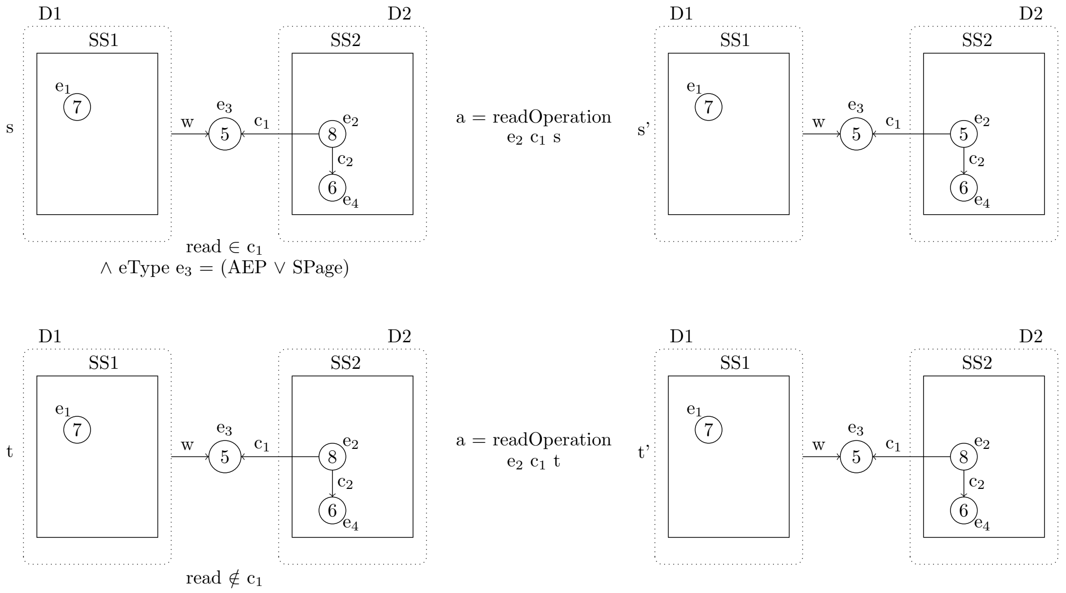

Figure 7.14: Noninterference for Read on object types = Asynchronous IPC Endpoint executed from Domain 2

Preconditions:

*	$s \overset{D1}{\sim} t \equiv$ 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

Proof of the noninterference property for `read` on AEP or SPage objects from Domain 2:

For all e ∈ D1, we have:

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

Hence aquiv_nonin s' t' D1 , what is equivalent to $s' \overset{D1}{\sim} t'$

With $s' \overset{D1}{\sim} t'$ the noninterference property for `write` on AEP or SPage objects from Domain 1 is fulfilled.

7.5 Remove

`Remove` can be executed on CNode, VSpace or Interrupt Controller object types.
As in previous chapters 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 a 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 or SPage 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 runs as illustrated in figure 7.15.

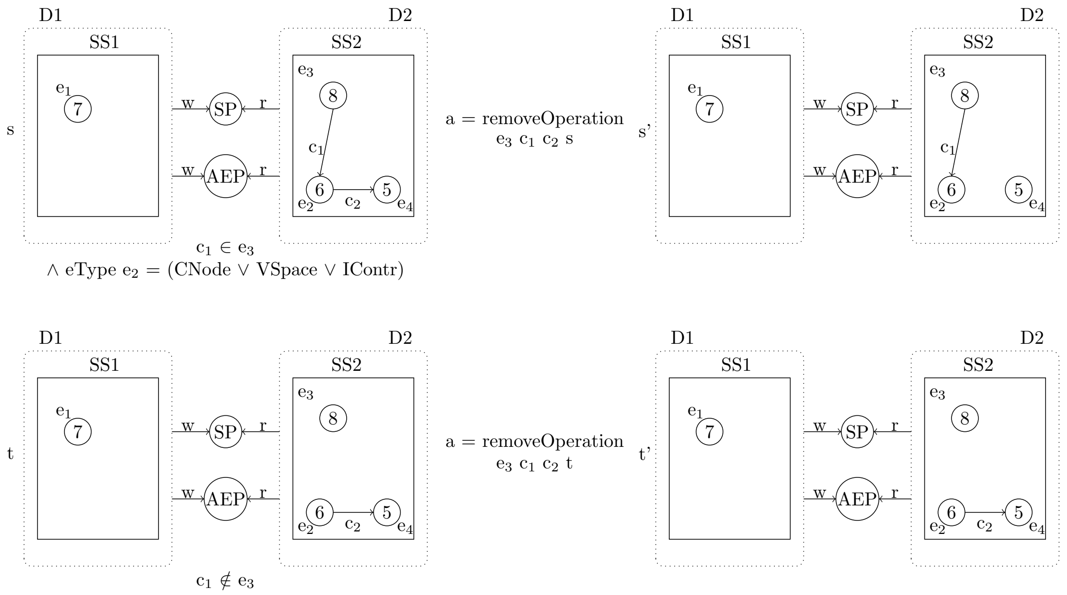

Figure 7.15: Noninterference for Remove on object types = CNode, VSpace or IContr. The removed capability points to an entity in the same domain

Preconditions:

* $s \overset{D1}{\sim} t \equiv$ aquiv_nonin s t D1

** removeOperation e_2 c_1 c_2 s removes a capability $\in e_2 \notin D1$ that points on an entity $\notin D1$.
That means it has no impact on any entity $\in D1$

*** legal (SysRemove e_2 c_1 c_2) t = false $\Rightarrow$ t' = t

Proof of the noninterference property for `remove` on CNode, VSpace or IContr objects (removed capability points on an entity in the same domain):

For all e ∈ D1, we have:

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

Hence aquiv_nonin s' t' D1 , what is equivalent to s' $\overset{D1}{\sim}$ t'

With s' $\overset{D1}{\sim}$ t' the noninterference property for `remove` on CNode, VSpace or IContr objects where the removed capability points on an entity in the same domain is fulfilled.

- Target object = AEP or SPage object
  If the removed capability points to an AEP or SPage object, it may be possible that information flows out of the `H` domain to the `L` domain. Figure 7.16 displays that no information flowes to domain 1.

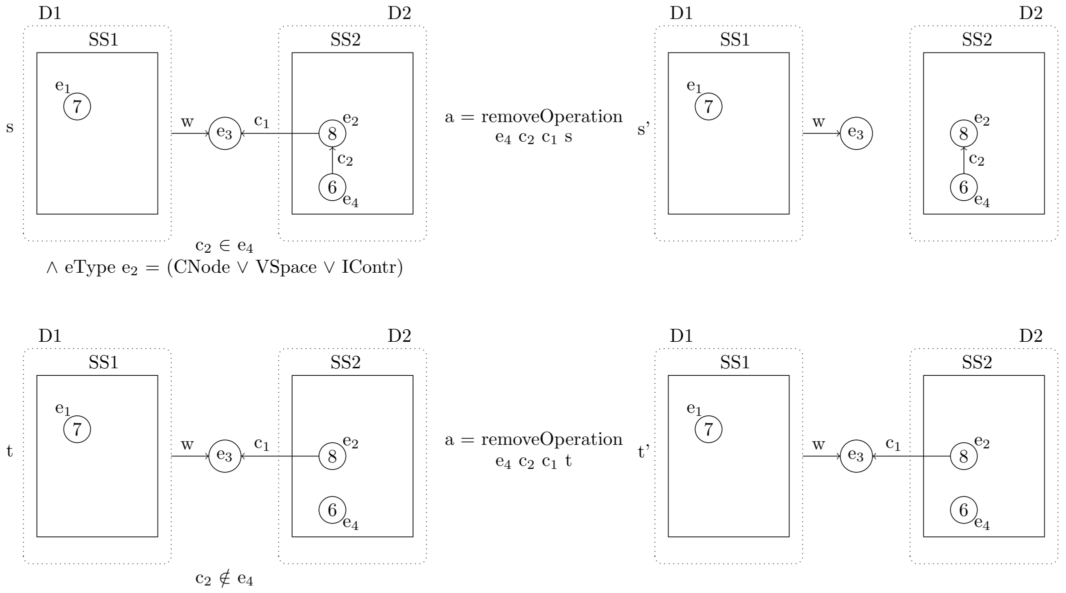

Figure 7.16: Noninterference for Remove on object types = CNode, VSpace or IContr. The removed capability points to an entity outside the H domain

Preconditions:

* s $\overset{D1}{\sim}$ t ≡ aquiv_nonin s t D1

** removeOperation e_2 c_1 c_2 s removes a capability ∈ e_2 ∉ D1 that points on an entity ∉ D1. This entity also owns no capability that points on an entity ∈ D1. That means it has no impact on any entity ∈ D1

*** legal (SysRemove e_2 c_1 c_2) t = false ⇒ t' = t

Proof of the noninterference property for `remove` on CNode, VSpace or IContr objects where the removed capability points on an object = AEP ∨ SPage:

For all e ∈ D1, we have:

$$(\text{value\_of s' e} \overset{**}{=} \text{value\_of s e} \overset{*}{=} \text{value\_of t e} \overset{***}{=} \text{value\_of t' e}$$
$$\land \quad \text{caps\_of s' e} \overset{**}{=} \text{caps\_of s e} \overset{*}{=} \text{caps\_of t e} \overset{***}{=} \text{caps\_of t' e}$$
$$\land \quad \text{subSys s' e} \overset{**}{=} \text{subSys s e} \overset{*}{=} \text{subSys t e} \overset{***}{=} \text{subSys t' e})$$

Hence aquiv_nonin s' t' D1 , what is equivalent to s' $\overset{D1}{\sim}$ t'

With s' $\overset{D1}{\sim}$ t' the noninterference property for `remove` on CNode, VSpace or IContr objects where the removed capability points on an entity in the same domain is fulfilled.

7.5.2 Remove on objects ≠ CNode, VSpace and Interrupt Controller

On all other object types the execution of `remove` is not legal. But for the sake of completeness I consider the difference between a target object of the removed capability in the executing domain and one outside.

- Target object of the removed capability is in the same domain
  The execution is not legal because the object on which the operation is executed ≠ CNode, VSpace and Interrupt Controller.

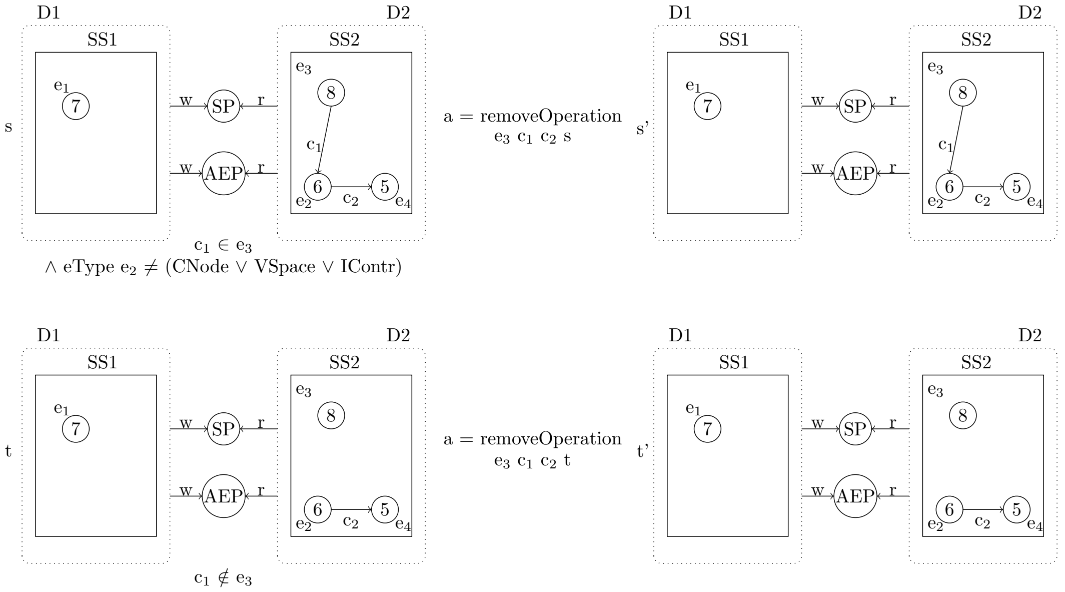

Figure 7.17: Noninterference for Remove on objects ≠ CNode, VSpace and ICont. The removed capability points to an entity in the same domain

Preconditions:

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

Proof of the noninterference property for `remove` on objects ≠ CNode,

VSpace and IContr where the removed capability points on an entity in the same domain:

For all e ∈ D1, we have:

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

Hence aquiv_nonin s' t' D1 , what is equivalent to s' $\overset{D1}{\sim}$ t'

With s' $\overset{D1}{\sim}$ t' the noninterference property for `remove` on objects ≠ CNode, VSpace and ICont where the removed capability points on an entity in the same domain is fulfilled.

- Target object = AEP or SPage
  If the removed capability points to an AEP or SPage object, also nothing happens to Domain 1 as the execution is not legal.

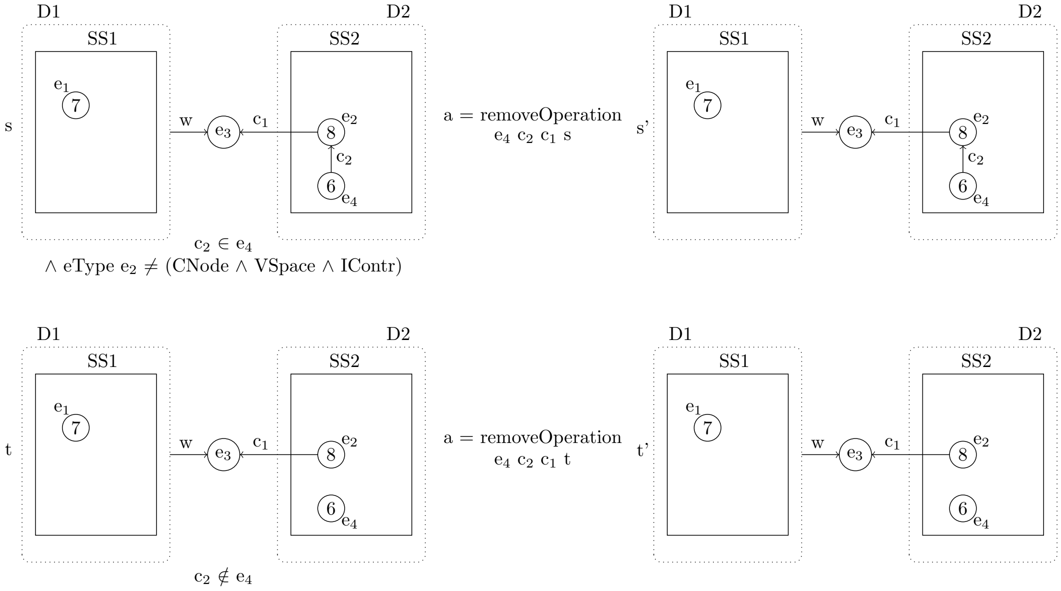

Figure 7.18: Noninterference for Remove on object types ≠ CNode, VSpace or IContr. The removed capability points to an entity outside the H domain

Preconditions:

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

Proof of the noninterference property for `remove` on objects ≠ CNode, VSpace or IContr where the removed capability points on an object = AEP ∨ SPage:

For all e ∈ D1, we have:

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

Hence aquiv_nonin s' t' D1 , what is equivalent to s' $\overset{D1}{\sim}$ t'

With s' $\overset{D1}{\sim}$ t' the noninterference property for `remove` on objects ≠ CNode, VSpace or IContr where the removed capability points on an entity in the same domain is fulfilled.

7.6 Revoke

With `revoke` the authority of a whole subsystem can be removed. As mentioned in chapter 2.2.2 the kernel keeps a record of all capabilities in the system with a *Capability Derivation Tree* (CDT). When `SysRevoke e c s` is performed, all children of c in the CDT are deleted with the `remove` operation executed on each of them. `Revoke` is legal on CNode and Untyped Memory objects. Like in chapter 7.5 I divide the proof in 4 parts.

7.6.1 Revoke on CNode or Untyped Memory objects

Revoke deletes all children of a specified capability. They can point on entities in the same domain or on an AEP or SPage object.

- The targets of the deleted entities are in the same domain
  Figure 7.19 illustrates the run of an `revoke` operation, if the removed capabilities point on entities in the same domain and `revoke` is legal for the executed entity.

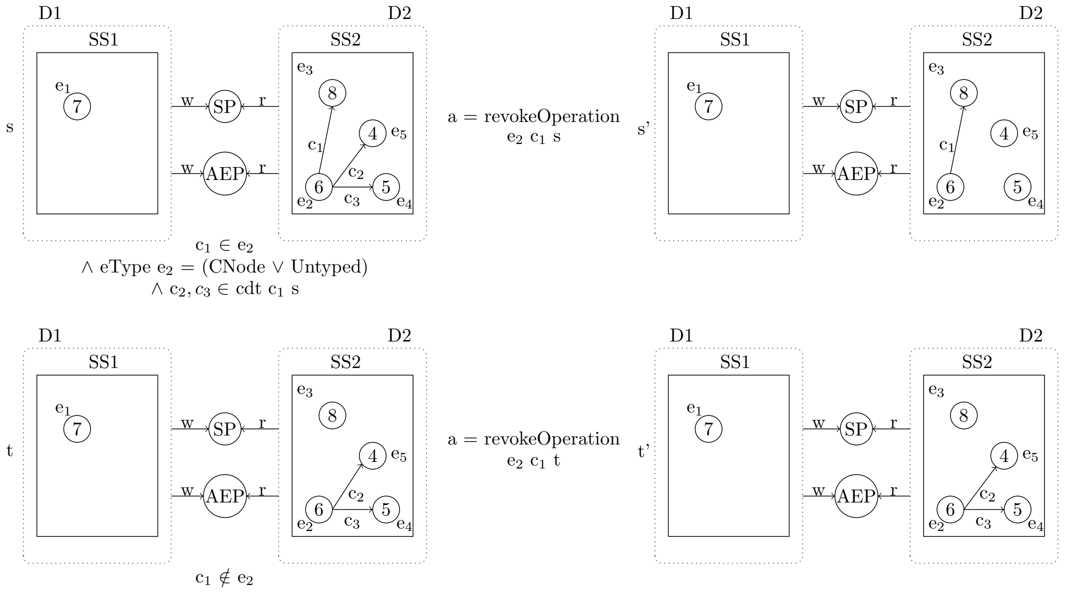
D1
D2
SS1
SS2
s
s'
t
t'
e1
7
e3
8
4
e5
6
e2
5
e4
c1
c2
c3
w
r
SP
AEP
a = revokeOperation e2 c1 s
a = revokeOperation e2 c1 t
c1 ∈ e2
∧ eType e2 = (CNode ∨ Untyped)
∧ c2, c3 ∈ cdt c1 s
c1 ∉ e2

Figure 7.19: Noninterference for Revoke on object types = CNode or Untyped. The removed capabilities point on entities in the same domain

Preconditions:

* $s \overset{D1}{\sim} t \equiv$ aquiv_nonin s t D1

** revokeOperation e_2 c_1 s removes capabilities ∈ entities ∉ D1 that point on entities ∉ D1.
That means it has no impact on any entity ∈ D1

*** legal (SysRevoke e_2 c_1) t = false ⇒ t' = t

Proof of the noninterference property for `revoke` on CNode or Untyped objects where the removed capabilities point on entities in the same domain:

For all e ∈ D1, we have:

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

Hence aquiv_nonin s' t' D1 , what is equivalent to s' $\overset{D1}{\sim}$ t'

With s' $\overset{D1}{\sim}$ t' the noninterference property for `revoke` on CNode or Untyped objects, where the removed capabilities point on entities in the same domain, is fulfilled.

- Target object = AEP or SPage object
  Like in the `remove` operation I have to control if information flowes to Domain 1 if the removed capability points on an AEP or SPage object. Figure 7.16 illustrates it.

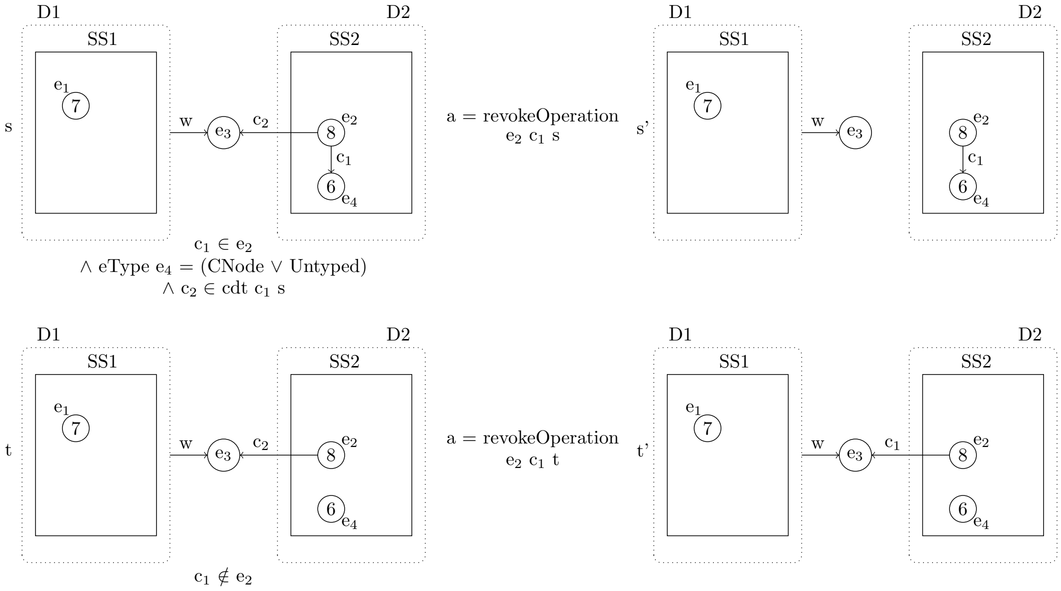

Figure 7.20: Noninterference for Revoke on object types = CNode or Untyped. The removed capabilities point on entities outside the H domain

Preconditions:

*	s $\overset{D1}{\sim}$ t ≡ aquiv_nonin s t D1
**	revokeOperation e_2 c_1 s removes capabilities ∈ entities ∉ D1 that point on entities ∉ D1. This entities also own no capability that points on an entity ∈ D1. That means it has no impact on any entity ∈ D1
***	legal (SysRevoke e_2 c_1) t = false ⇒ t' = t

Proof of the noninterference property for `remove` on CNode or Untyped objects, where the removed capabilities point on objects = AEP ∨ SPage:

For all e ∈ D1, we have:

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

Hence aquiv_nonin s' t' D1 , what is equivalent to s' $\overset{D1}{\sim}$ t'

With s' $\overset{D1}{\sim}$ t' the noninterference property for `remove` on CNode or Untyped objects, where the removed capabilities point on objects = AEP ∨ SPage, is fulfilled.

7.6.2 Revoke on objects ≠ CNode and Untyped Memory

On all other object types the execution of `revoke` is not legal. Again I differ between target objects, of the removed capabilities, in the executing domain and those outside.

- Target objects of the removed capabilities are in the same domain
  The execution is not legal because the object the operation is executed on ≠ CNode and Untyped Memory.

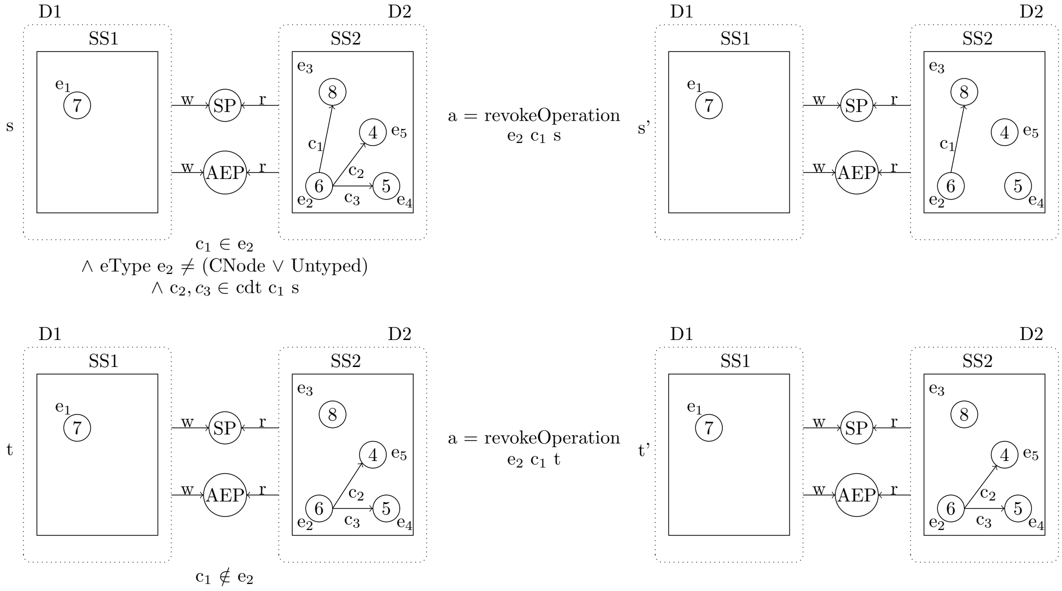

Figure 7.21: Noninterference for Revoke on objects ≠ CNodeand Untyped Memory. The removed capabilities point on entities in the same domain

Preconditions:

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

Proof of the noninterference property for `revoke` on objects ≠ CNode and Untyped where the removed capabilities point on entities in the same domain:

For all e ∈ D1, we have:

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

Hence aquiv_nonin s' t' D1 , what is equivalent to s' $\overset{D1}{\sim}$ t'

With s' $\overset{D1}{\sim}$ t' the noninterference property for `revoke` on objects ≠ CNode and Untyped, where the removed capabilities point on entities in the same domain, is fulfilled.

- Target objects = AEP or SPage

  If the removed capabilities point on AEP or SPage objects, nothing happens either to Domain 1 as the execution is not legal.

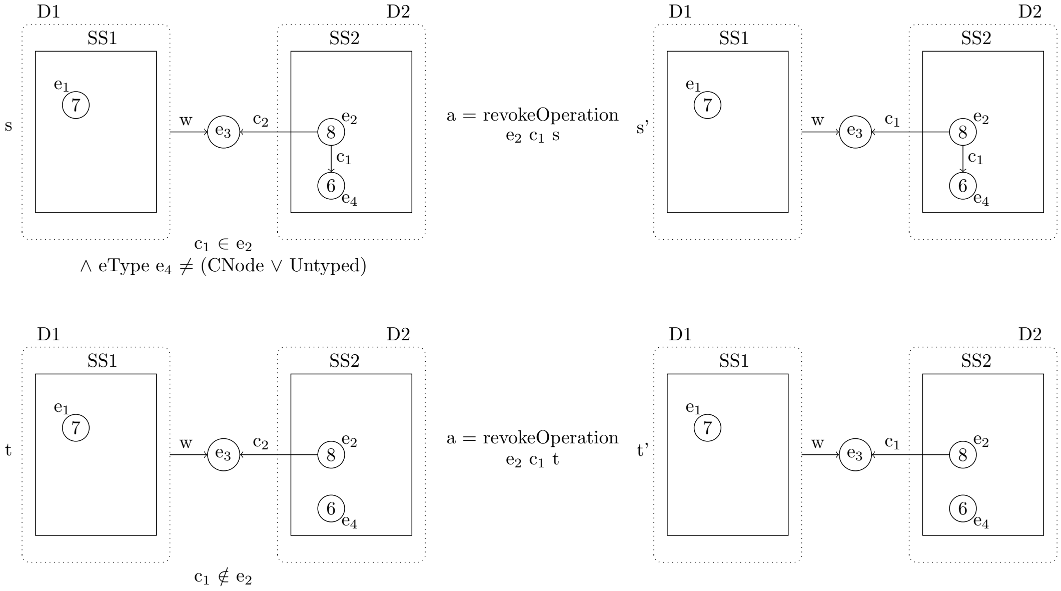

Figure 7.22: Noninterference for Revoke on object types ≠ CNode and Untyped where the removed capabilities point on entities outside the H domain

Preconditions:

$*$ s $\overset{D1}{\sim}$ t $\equiv$ aquiv_nonin s t D1

$**$ legal (SysRevoke e_2 c_1) s = false $\Rightarrow$ s' = s

$***$ legal (SysRevoke e_2 c_1) t = false $\Rightarrow$ t' = t

Proof of the noninterference property for `revoke` on objects $\neq$ CNode and Untyped, where the removed capabilities point on objects = AEP $\vee$ SPage:

For all e $\in$ D1, we have:

(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)

Hence aquiv_nonin s' t' D1 , what is equivalent to s' $\overset{D1}{\sim}$ t'

With s' $\overset{D1}{\sim}$ t' the noninterference property for `revoke` on objects $\neq$ CNode and Untyped, where the removed capabilities point on objects = AEP $\vee$ SPage is fulfilled.

8 Conclusion

Summarized I tried to show the noninterference property on the take-grant model as it was specified by the team of NICTA in the paper ”Verified Protection Model of the seL4 Microkernel”[1]. This attempt failed so I had to make the model more precise based on my alalysis in Chapter 5 of noninterference in the original model. That means I defined `read` and `write` operations, a value and object type for entities and a check if the object type is able to perform the particular system operation.
Theorem 1 in Chapter 5 formed the theory of this thesis. With the adaptions in Chapter 6 it was feasible to investigate *Theorem 1* is fulfilled for the system operations of the take-grant model.

The conclusion of the thesis is that the original model is not appropriate to show noninterference on it. With the extended one it was possible and every system operation fulfills it.

As a next step the noninterference property should be specified and verified formally for the extended model. This can also be done with the theorem proof assistant Isabelle/HOL.

Bibliography

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

[2] 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

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

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

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

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

[7] 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

[8] R. J. Lipton and L. Snyder:
"A Linear Time Algorithm for Deciding Subject Security".
1977